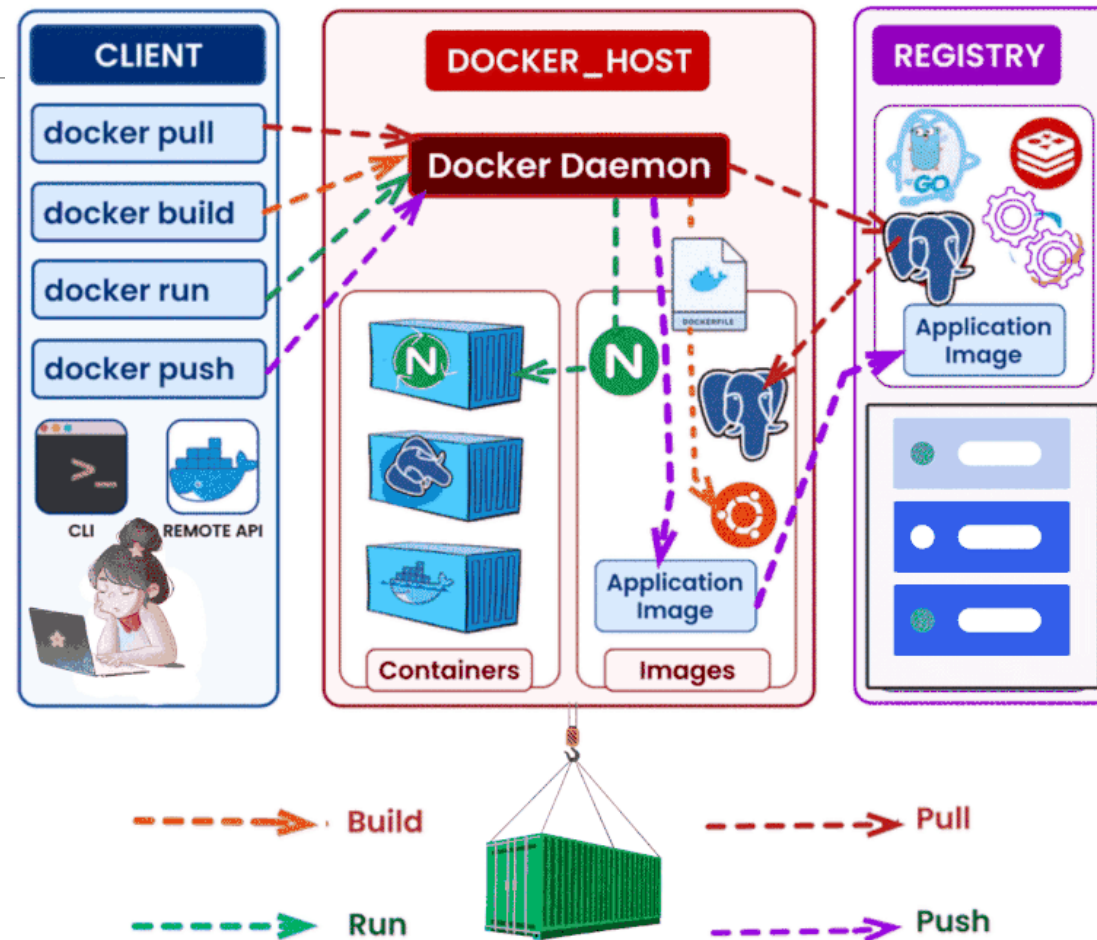


Paradigmas de Linguagem de Programação

DOCKER ARCHITECTURE



Vamos criar nossas imagens e subir nossos microservices em containers?

MAS ANTES, VAMOS TEMOS QUE PREPARAR O AMBIENTE

Primeiro Passo: Compilar e Empacotar o Microservice

Nosso projeto usa o **Maven** (gerenciador de build e dependências do Java).

👉 Vamos utilizar o **Maven Wrapper (mvnw)** , que permite executar Maven **sem precisar instalá-lo** na máquina: `./mvnw clean package -DskipTests`

O que é o Maven Wrapper?

- É um conjunto de scripts e arquivos que:
 - Garante o uso de uma **versão específica do Maven**
 - Permite compilar o projeto sem depender do Maven instalado localmente
 - Arquivos típicos:
 - mvnw, mvnw.cmd
 - .mvn/wrapper/*
- ✅ Ideal para times, CI/CD e ambientes padronizados

Primeiro Passo: Compilar e Empacotar o Microservice



Fases principais do ciclo de vida do Maven

<code>mvn clean</code>	→ apenas limpa o build anterior
<code>mvn compile</code>	→ compila o código (sem empacotar)
<code>mvn test</code>	→ compila e executa os testes
<code>mvn package</code>	→ compila, testa e empacota (geralmente jar/war)
<code>mvn install</code>	→ faz tudo até o package e instala no repositório local
<code>mvn deploy</code>	→ faz tudo e envia para repositório remoto

Primeiro Passo: Compilar e Empacotar o Microservice

Para compilar e empacotar **sem rodar os testes**, usamos:

```
./mvnw clean package -DskipTests
```

 Isso é útil quando:

- Os testes já foram executados anteriormente
- Estamos apenas preparando o **.jar** para subir em container

Segundo Passo: Preparar os arquivos para Imagem Docker

Para subir os microservices como containers, precisamos criar **imagens Docker**.

👉 Para isso, usamos:

`Dockerfile`: descreve como a imagem será construída

`.dockerignore`: evita copiar arquivos desnecessários para a imagem

Segundo Passo:

Preparar os arquivos para Imagem Docker

O que é o Dockerfile?

Arquivo de instruções que define como o Docker vai montar a imagem.

◆ Exemplo:

```
FROM eclipse-temurin:21-jre-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Esse exemplo usa uma imagem leve com Java 21 e copia o **.jar** gerado para dentro da imagem

Segundo Passo:

Preparar os arquivos para Imagem Docker

O que é o `.dockerignore`?

Evita que arquivos desnecessários sejam enviados para o build da imagem.

- ◆ Funciona como um **`.gitignore`**, mas para o Docker.





- ◆ Exemplo:

```
target/  
!target/*.jar  
.git/  
.idea/  
*.iml  
*.md  
*.log
```

- ✓ Resultado: build mais rápido e imagem menor.

Segundo Passo: Preparar os arquivos para Imagem Docker

Por que usar `.dockerignore`?

-  Evita copiar arquivos grandes ou inúteis para dentro da imagem
-  Reduz o tamanho final da imagem
-  Acelera o processo de build
-  Evita vazamento de informações sensíveis (como chaves, configs locais)

Terceiro Passo:

Construir o `docker-compose.yml`

Depois de:

1. **Compilar e empacotar** os microservices com Maven (`.mvnw clean package -DskipTests`)
2. Criar o `Dockerfile` e o `.dockerignore` para cada microservice
3. Agora o **passo 3 é montar o** `docker-compose.yml` para orquestrar os containers

Terceiro Passo:

Construir o `docker-compose.yml`

O que o `docker-compose.yml` vai fazer?

Subir os serviços juntos, com dependências como bancos de dados

Criar uma rede interna entre os containers (para comunicação interna)

Mapear as portas externas (ex: 8761 do Eureka, 8765 do Gateway)

Definir variáveis de ambiente, como URLs, credenciais, eureka, etc.

Buildar a imagem de cada serviço a partir do Dockerfile

```
1  ∨ services:
2
3      # -----
4      ▷ Run Service
5  → discovery-service:
6  ∨      image: discovery-service:latest
7      build:
8  ∨      context: ./discovery-service
9      ports:
10 ∨      - "8761:8761"
11      networks:
12      - spring-net
```

```
14  ✓ db-currency:
15      image: postgres:16
16  ✓ environment:
17      POSTGRES_DB: db_currency
18      POSTGRES_USER: postgres
19      POSTGRES_PASSWORD: postgres
20  ✓ networks:
21      - spring-net
```

```
23  ✓ currency-service:
24      image: currency-service:latest
25  ✓ build:
26      context: ./currency-service
27  ✓ depends_on:
28      - discovery-service
29      - db-currency
30  ✓ environment:
31      SPRING_DATASOURCE_URL: jdbc:postgresql://db-currency:5432/db_currency
32      SPRING_DATASOURCE_USERNAME: postgres
33      SPRING_DATASOURCE_PASSWORD: postgres
34      EUREKA_CLIENT_SERVICE_URL_DEFAULTZONE: http://discovery-service:8761/eureka
35  ✓ networks:
36      - spring-net
```

```
39 db-product:
40   image: postgres:16
41   environment:
42     POSTGRES_DB: db_product
43     POSTGRES_USER: postgres
44     POSTGRES_PASSWORD: postgres
45   networks:
46     - spring-net
```

▷ Run Service

```
47 product-service:
48   image: product-service:latest
49   build:
50     context: ./product-service
51   depends_on:
52     - discovery-service
53     - db-product
54     - currency-service
55   environment:
56     SPRING_DATASOURCE_URL: jdbc:postgresql://db-product:5432/db_product
57     SPRING_DATASOURCE_USERNAME: postgres
58     SPRING_DATASOURCE_PASSWORD: postgres
59     EUREKA_CLIENT_SERVICE_URL_DEFAULTZONE: http://discovery-service:8761/eureka
60   networks:
61     - spring-net
```

```
64  ✓ gateway-service:
65      image: gateway-service:latest
66  ✓ build:
67      context: ./gateway-service
68  ✓ ports:
69      - "8765:8765"
70  ✓ depends_on:
71      - discovery-service
72  ✓ environment:
73      EUREKA_CLIENT_SERVICE_URL_DEFAULTZONE: http://discovery-service:8761/eureka
74  ✓ networks:
75      - spring-net
```

```
78 networks:
79     spring-net:
80         driver: bridge
81
```


Outra opção para a compilação e criação da imagem Docker

MULTI-STAGE BUILD

Multi-stage Build no Docker

Build totalmente automatizado

- Você **não precisa compilar manualmente** antes de usar o `docker image build`

Reprodutibilidade

- O build é sempre feito de forma **padronizada e consistente**.
- Se alguém clonar seu projeto e der `docker image build`, vai obter o mesmo resultado, sem depender do Maven ou Java na máquina local.

Ideal para CI/CD

- No CI (como GitHub Actions, GitLab CI, etc.), você pode simplesmente fazer `docker image build` sem etapas extras.
- Isso facilita muito a automação de builds e deploys.

Multi-stage Build no Docker

```
# Etapa de build
FROM maven:3.9.9-eclipse-temurin-21-alpine AS build
WORKDIR /app
# Copia apenas o que é necessário para o build
COPY pom.xml .
COPY src ./src
# Realiza o build
RUN mvn clean package -DskipTests
# Etapa de runtime com imagem minimalista
FROM eclipse-temurin:21-jre
# Apenas o .jar final
COPY --from=build /app/target/*.jar /app/app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

```
##### Etapa de compilação da aplicação
FROM maven:3.9.9-eclipse-temurin-21-alpine AS build
WORKDIR /app
##### Copia apenas o que é necessário para a compilação
COPY pom.xml .
COPY src ./src
##### Realiza a compilação
RUN mvn clean package -DskipTests

##### Etapa de runtime com imagem minimalista
FROM eclipse-temurin:21-jre-alpine
VOLUME /tmp
COPY --from=build /app/target/*.jar /app/app.jar
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```