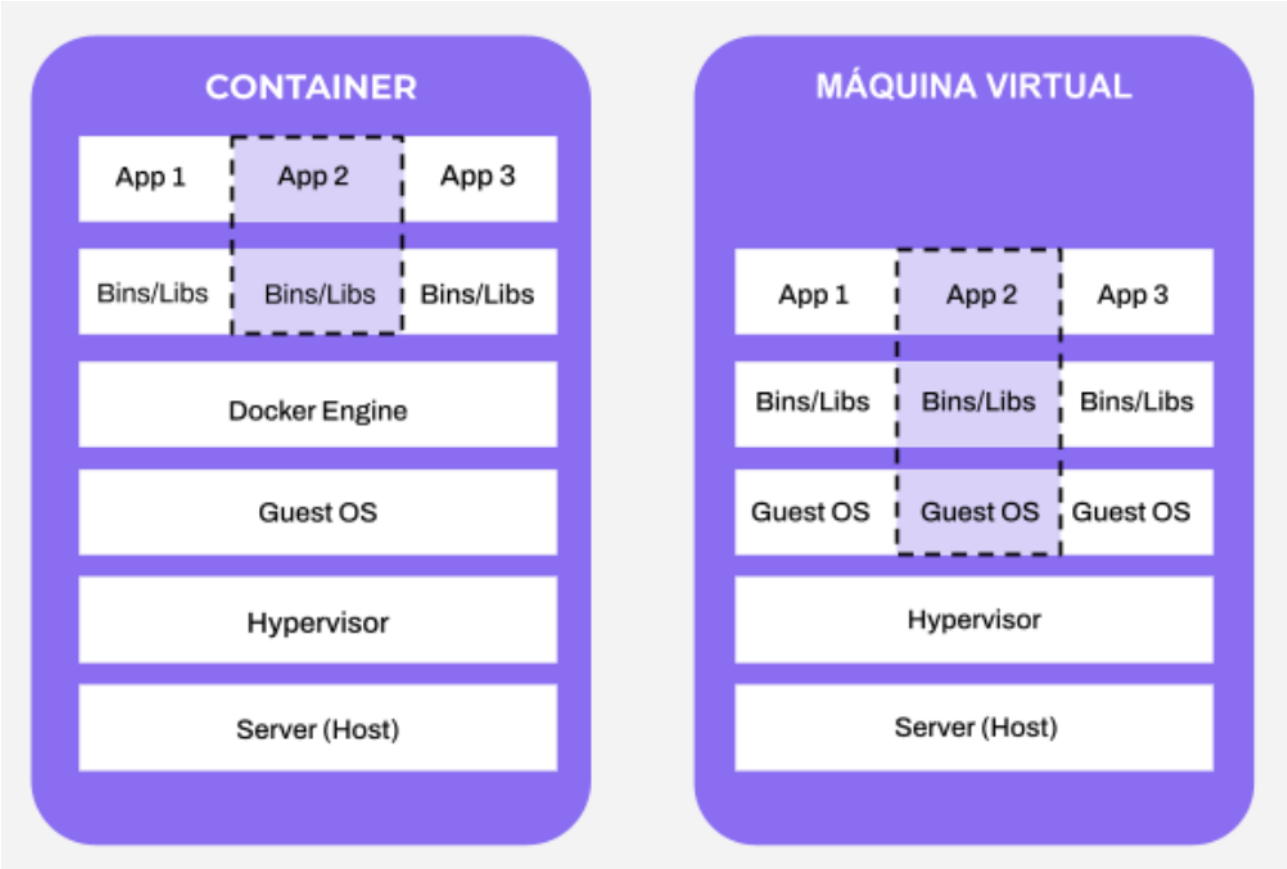
Docker

O Docker tornou o conceito de recipientes muito popular como uma alternativa leve às máquinas virtuais em 2013. Recapitulando: Docker nos fornece uma maneira padronizada de executar nosso código. Semelhante a como uma máquina virtual virtualiza o hardware do servidor, os containers virtualizam o sistema operacional de um servidor. Ao contrário de uma máquina virtual que usa um hipervisor para executar uma cópia completa de um sistema operacional em cada máquina virtual, os recipientes não fazem uma cópia completa de uma máquina, mas virtualizam as camadas de software acima do sistema operacional.



Por que usar o Docker?

O Docker nos permite padronizar os ambientes de implantação. Isto significa que um terceiro pode replicar um ambiente a partir de um roteiro, reduzindo o tempo de configuração e os problemas a zero. Por outro lado, sendo ambientes reduzidos, o uso dos recursos do servidor é otimizado, reduzindo os custos e aumentando a eficiência.

Ao mesmo tempo, sendo ambientes virtualizados, o tempo de início e reinício das instâncias é reduzido a praticamente zero. Tudo isso facilita a escalabilidade da solução. Basta executar uma nova imagem e você tem uma nova instância em um clique.

Na prática, existem scripts que copiam diretamente o código de uma pasta ou repo Git em uma Docker image pré-preparada, executam uma série de testes e comandos, implantam e iniciam automaticamente.

Deployando Eureka Server com Docker Compose

Para implantar o Eureka Server, devemos criar o dockerfile da mesma forma que com o resto dos microsserviços, mas mudamos a porta para 8761 ao invés de 8080. Ficaria da seguinte forma:

```
FROM adoptopenjdk/openjdk11:alpine-jre
ARG JAR_FILE=spring-boot-web.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","app.jar"]
EXPOSE 8761
```

No docker-compose.yml adicionamos:

```
eureka-server:
  build: spring-cloud/eureka-server
  mem_limit: 512m
  ports:
    - "8761:8761"
```

Modificações nas application.properties dos microsserviços

Quando executamos os microsserviços usando Docker, temos que modificar a URL de Eureka, pois ela não será encontrada na URL que definimos por default (http://localhost:8761/eureka/). Agora, em vez de localhost, teremos que colocar eureka-server, pois este é o nome que usamos no docker-compose.yml. Então ficaríamos com:

```
eureka.client.serviceUrl.defaultZone: http://eureka:8761/eureka/
```

Para tais casos, onde as propriedades mudam dependendo do ambiente, é bom usar profiles.

Abaixo, podemos ver o docker-compose.yml final com dois microsserviços, Eureka Server e MongoDB:

```
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"
  eureka-server:
    build: spring-cloud/eureka-server
    mem_limit: 512m
    ports:
      - "8761:8761"
 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
```

Conclusão

Como vimos nesta aula, nas arquiteturas de microsserviços temos rapidamente vários projetos para implantar. Também podemos ter vários bancos de dados de diferentes tipos, e gerenciar cada um desses componentes separadamente torna a manutenção muito difícil e demorada.

Docker Compose resolve este problema, permitindo-nos configurar em um arquivo YAML vários containers ao mesmo tempo. Então, com uma série de comandos, podemos compilá-los, executá-los e manipulá-los. Docker Compose é ideal para o desenvolvimento, testing e até mesmo fluxos contínuos de integração.