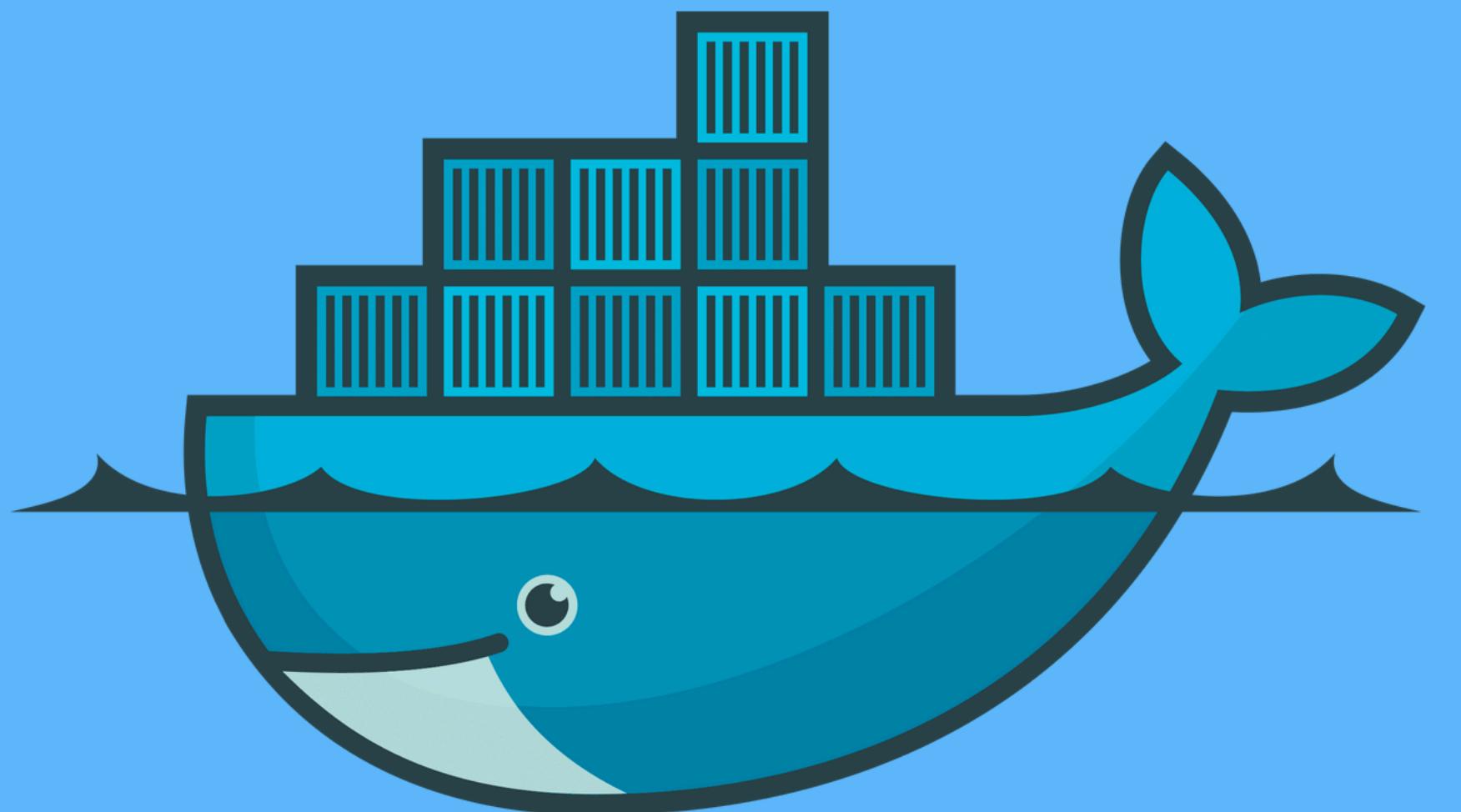


Trabalhando com
Docker



docker

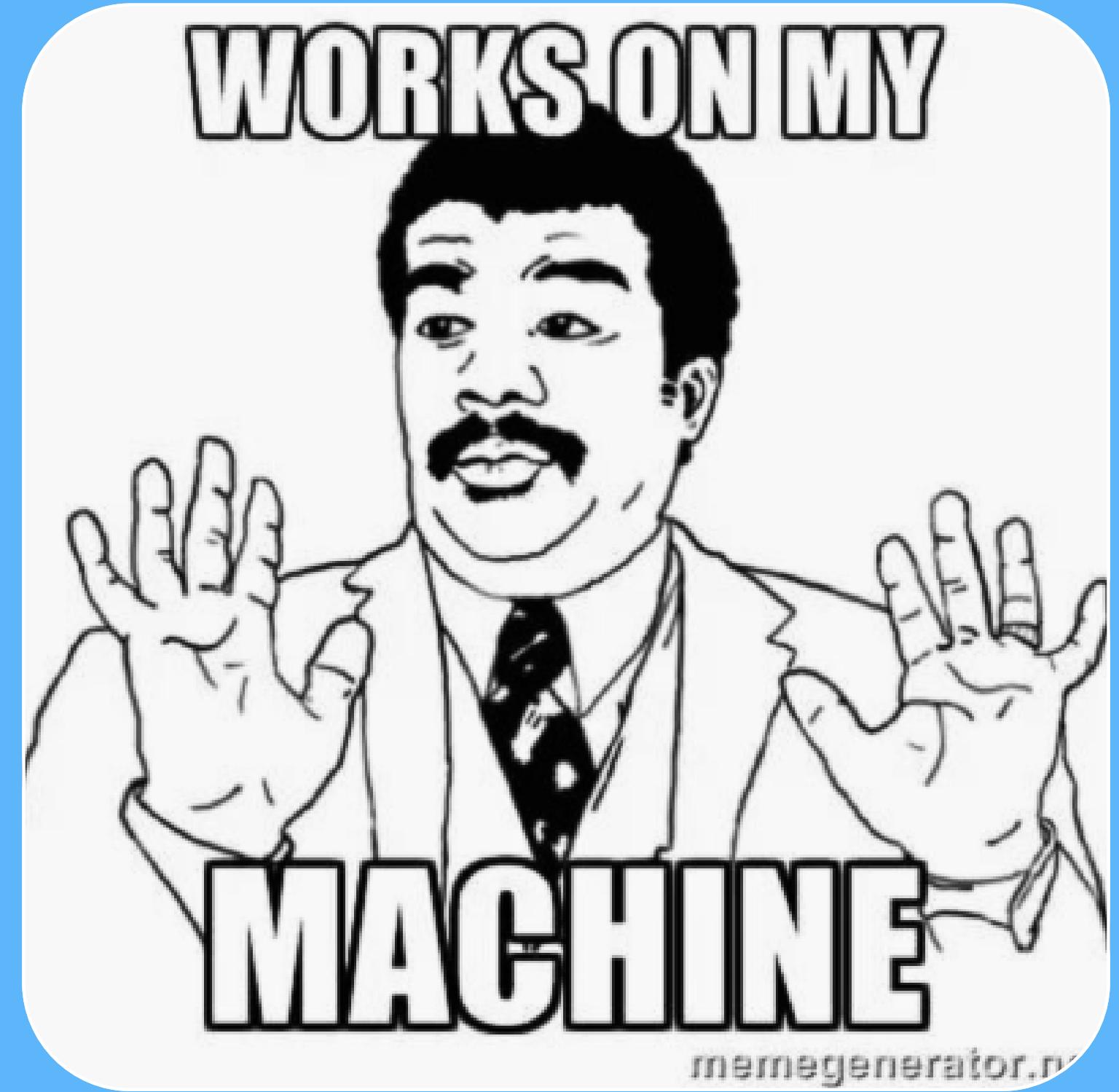
Agenda

O que é docker

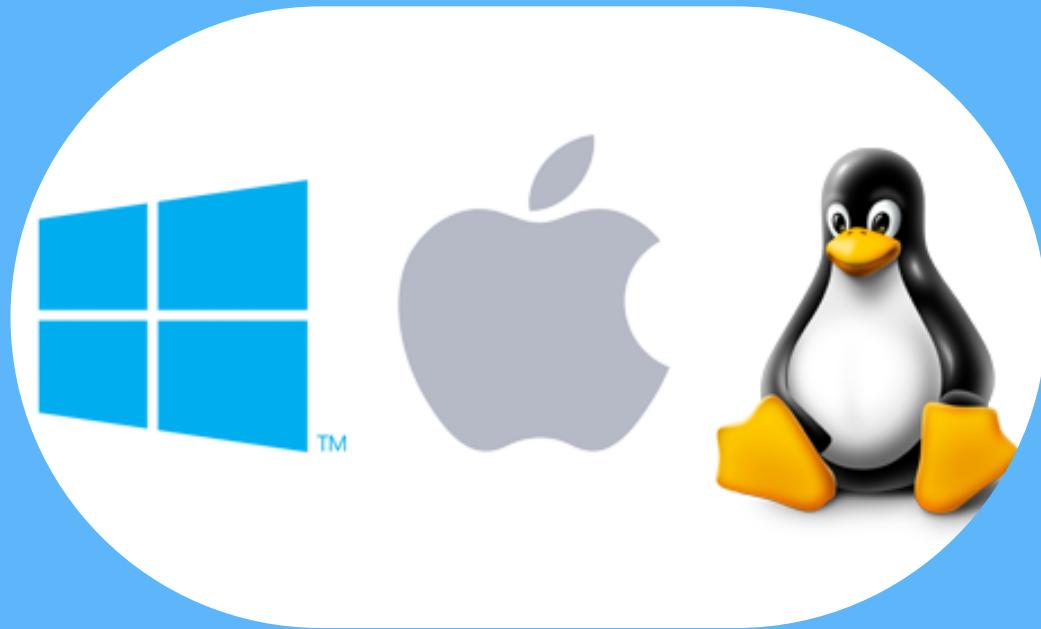
Images e Registry

Dockerfile e Docker Compose

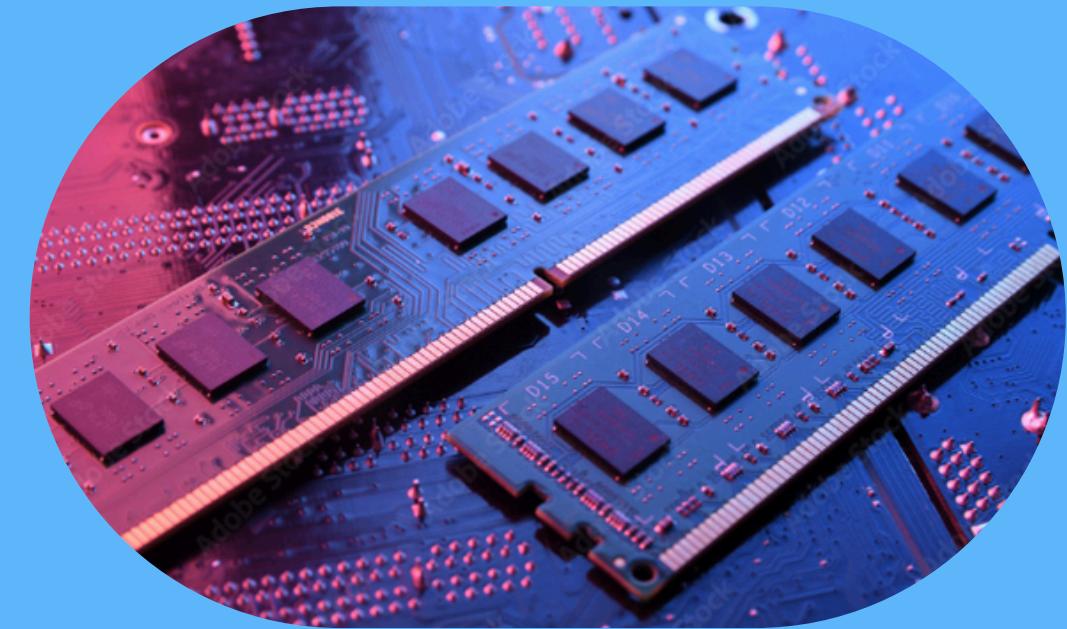
Por que usar docker?



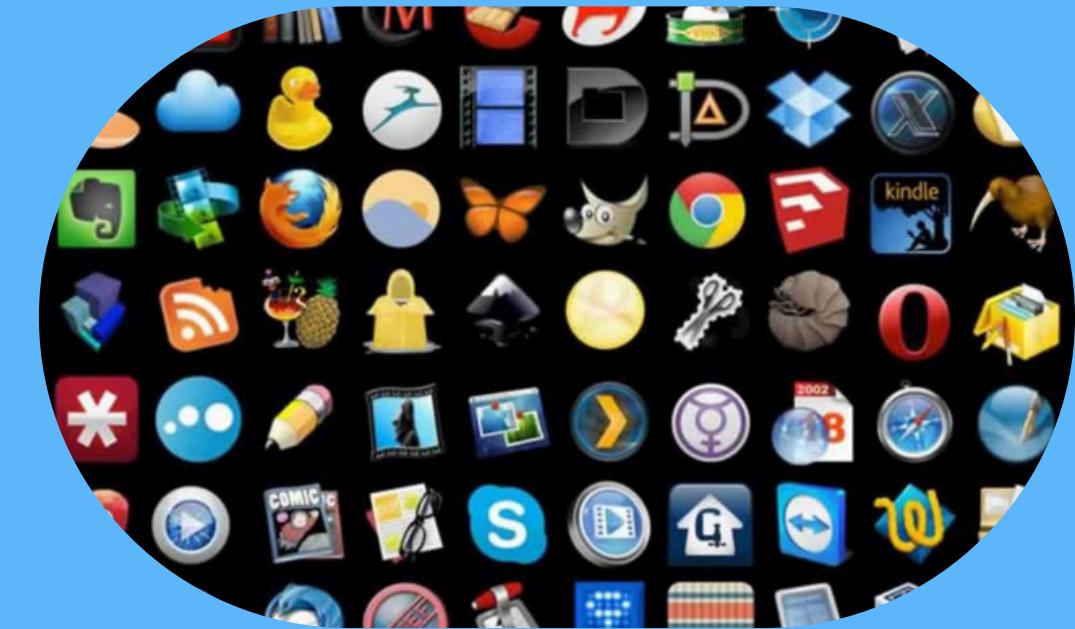
Como eram os tempos antes do Docker



Inconsistência entre ambientes



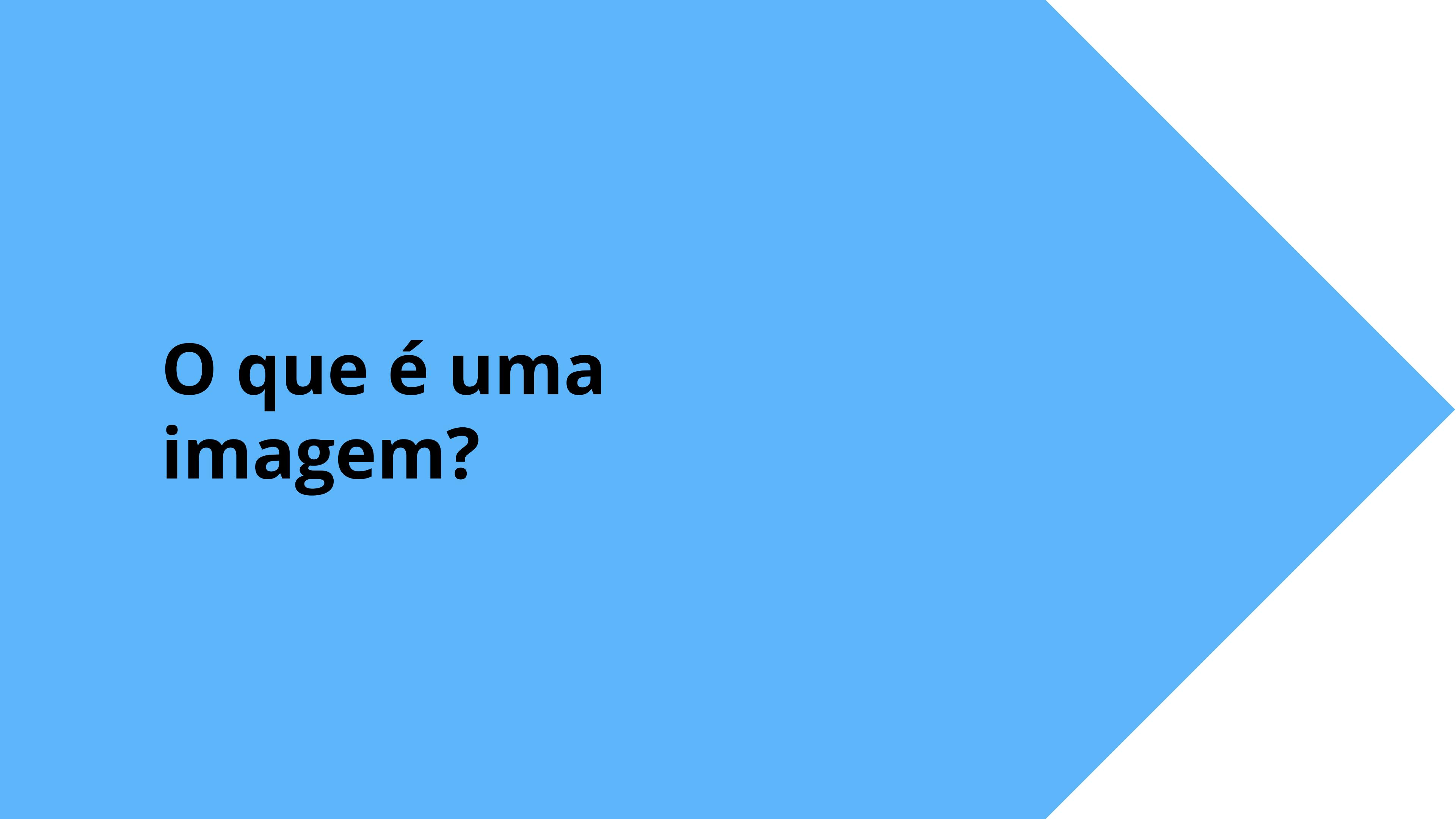
Ineficiência no uso de recursos



Ineficiência na configuração e
gestão de dependências e
serviços

O que é Docker?





O que é uma
imagem?

Uma pequena analogia



- Carros de fórmula 1
- Autódromo
- Arquibancada com platéia
- Publicidade

```
1 FROM golang:1.21-alpine AS builder
2
3 WORKDIR /app
4
5 COPY go.mod go.sum ./
```

```
6
7 RUN go mod download
8
9 COPY . .
10
11 RUN go build -o main cmd/app/main.go
12
13 FROM alpine:latest
14
15 WORKDIR /root/
16
17 COPY --from=builder /app/main .
18
19 EXPOSE 8080
20
21 CMD ["./main"]
```

```
1 FROM golang:1.21-alpine AS builder → Definição da imagem base  
2  
3 WORKDIR /app  
4  
5 COPY go.mod go.sum ./ → 1 module pricing-simulator  
6  
7 RUN go mod download 2  
8  
9 COPY . . 3 go 1.21.5  
10  
11 RUN go build -o main cmd/app/main.go 4  
12  
13 FROM alpine:latest 5 Check for upgrades | Upgrade transitive dependencies | Upgrade direct dependencies  
14  
15 WORKDIR /root/ 6 require github.com/gorilla/websocket v1.5.3 // indirect  
16  
17 COPY --from=builder /app/main .  
18  
19 EXPOSE 8080  
20  
21 CMD ["./main"]
```

Definição da imagem base utilizada

Empacotamento da aplicação com todas as suas dependências

Execução da imagem.

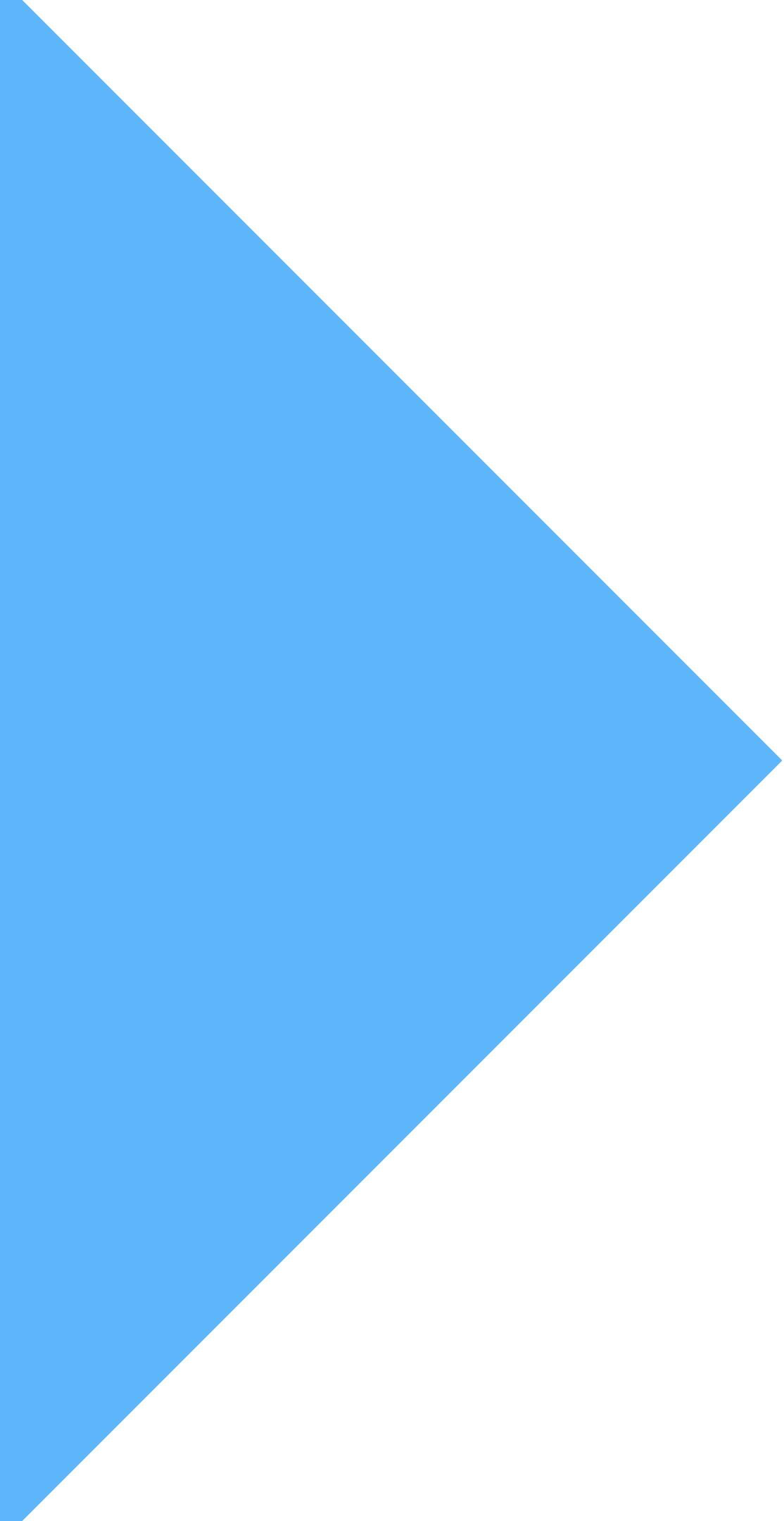
Para buildar a imagem descrita nesse arquivo basta executar o comando docker build .

O que é docker?



Imagen

É o empacotamento de uma aplicação (código) com todas as dependências necessárias para a sua execução.



O que é Docker registry?



É um repositório de imagens docker que pode ser local, público ou privado. Pense no docker registry como um bibliotecário que conhece todos os livros(imagens) registrados na biblioteca e que atua como um facilitador quando uma imagem é solicitada para compor um container.

O que é docker?

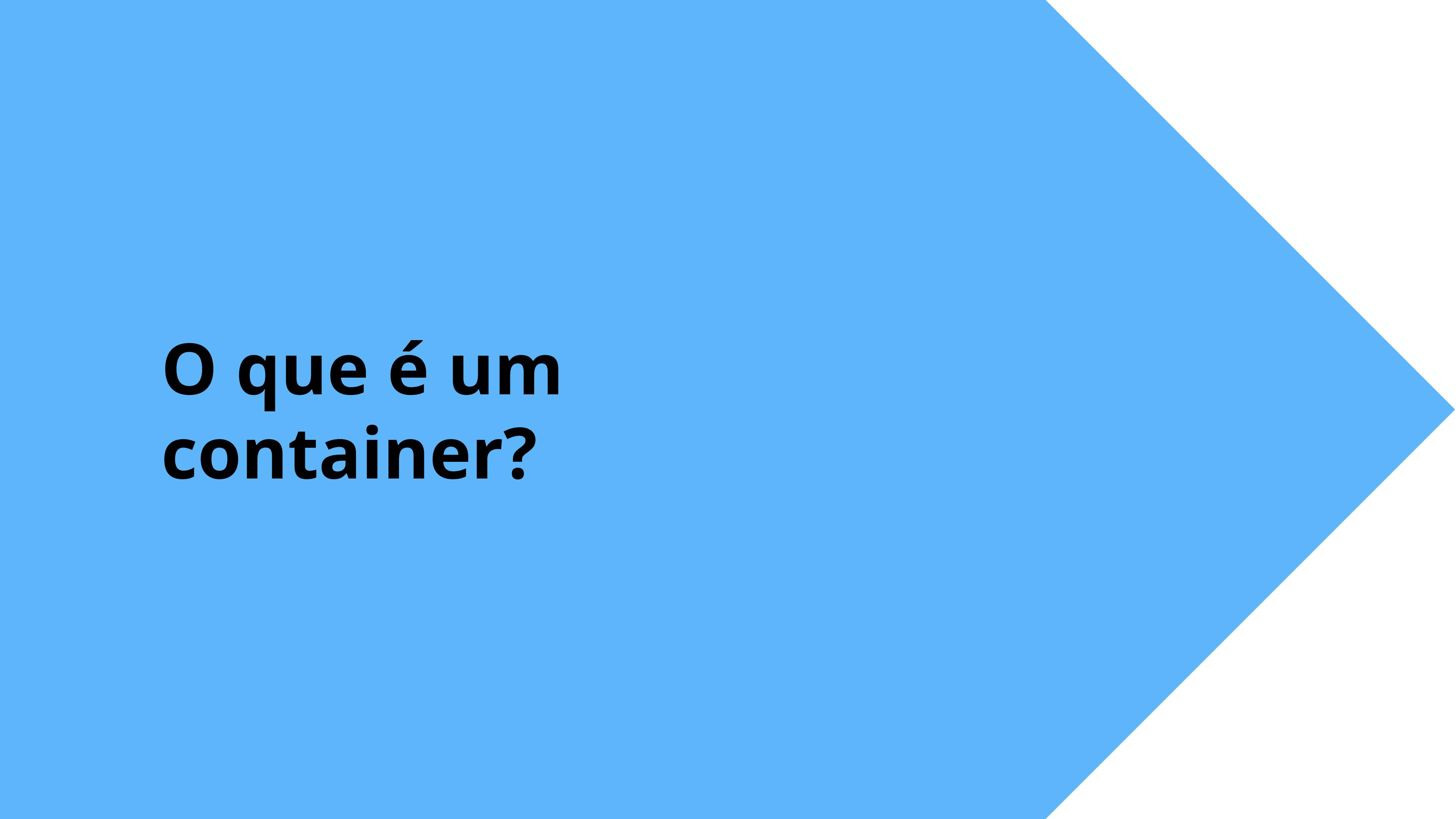


Imagen

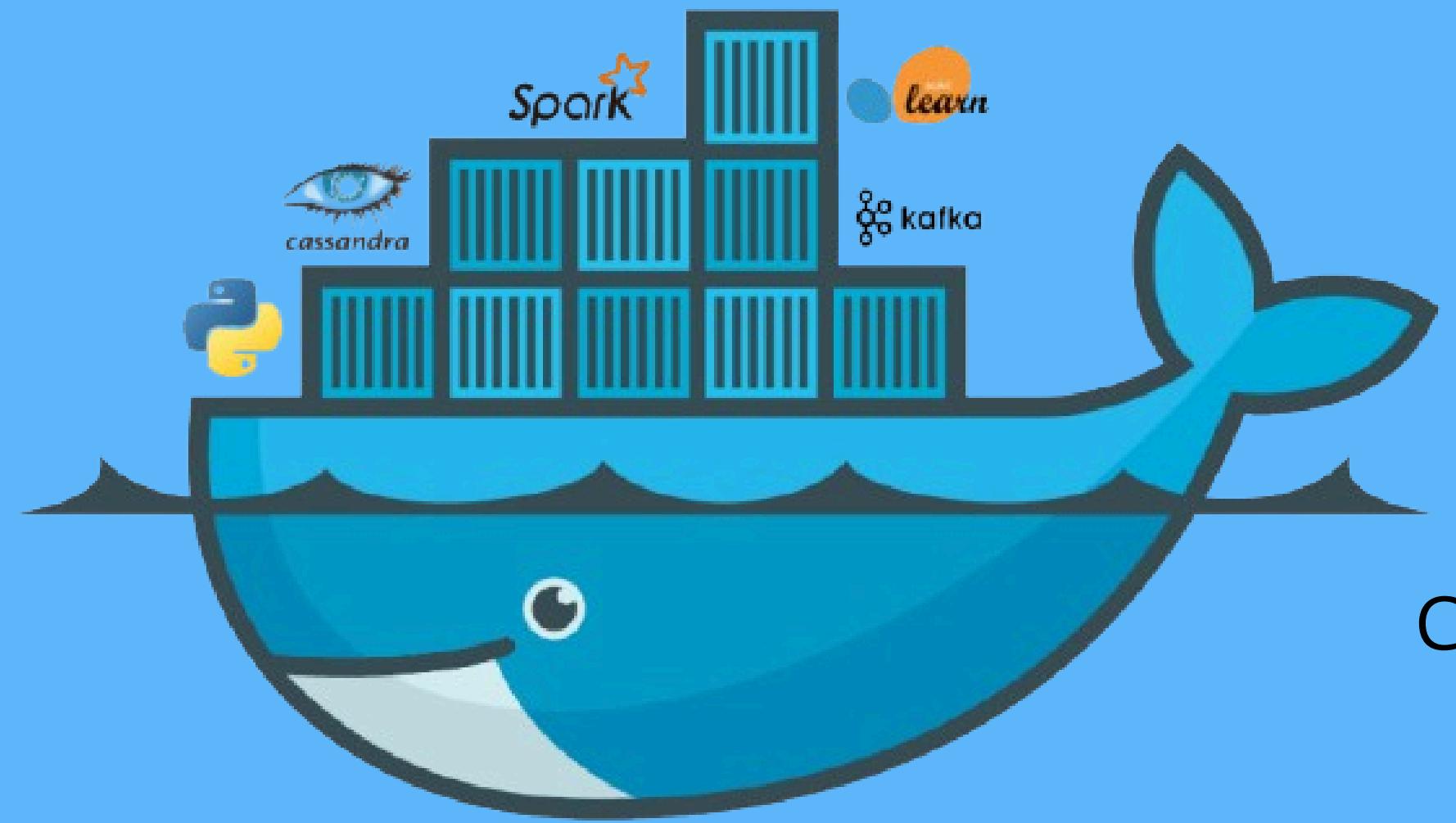
É o empacotamento de uma aplicação (código) com todas as dependências necessárias para a sua execução.

Registry

Um repositório de imagens que é utilizado como consulta de qual imagem utilizar para a construção de um container.



O que é um
container?



Container é a execução de uma imagem.

Container é a execução de uma imagem.

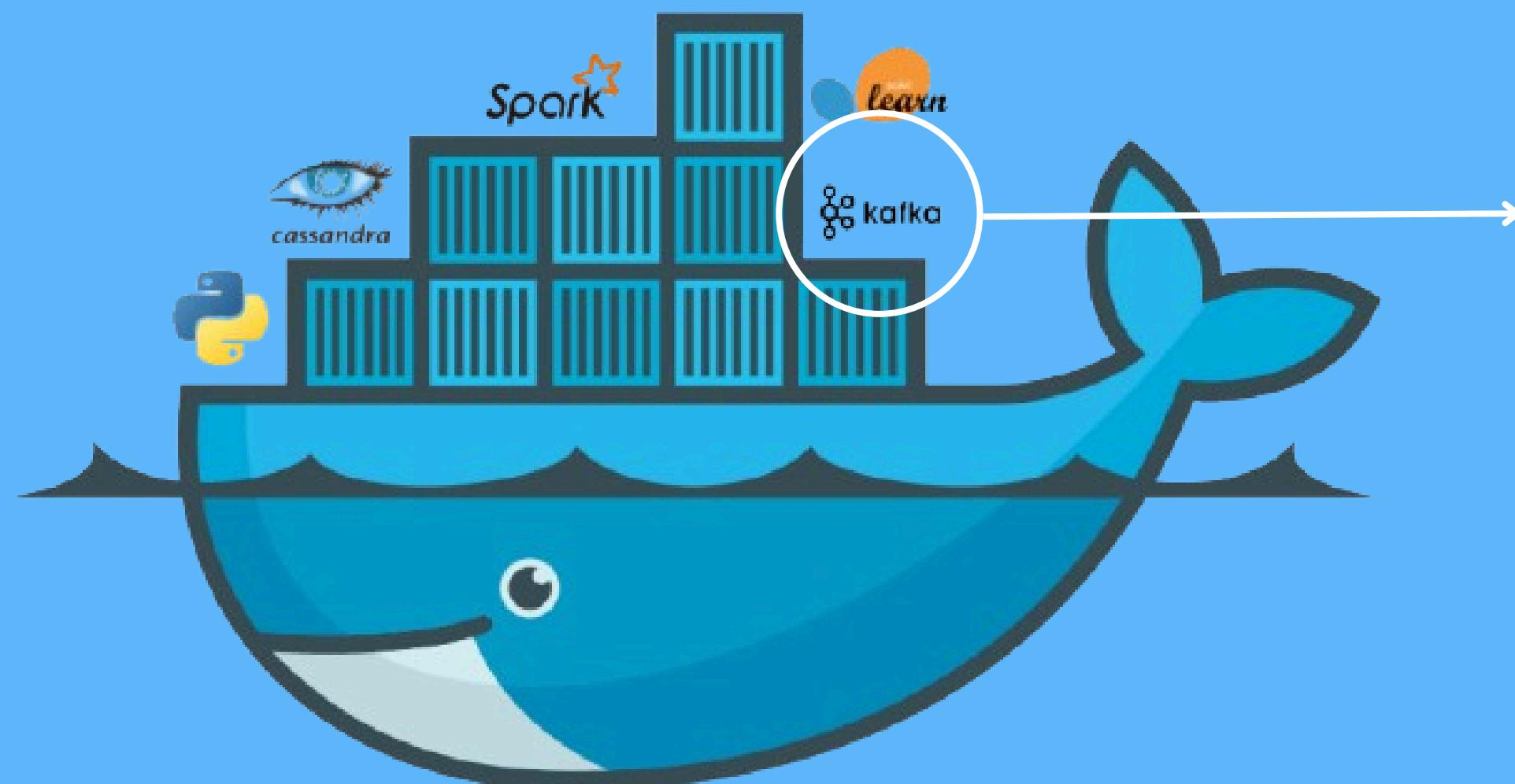


Image Layers	
1 ADD file ... in /	3.46 MB
2 CMD ["/bin/sh"]	0 B
3 ENV JAVA_HOME=/opt/java/openjdk	0 B
4 ENV PATH=/opt/java/openjdk/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin	0 B
5 ENV LANG=en_US.UTF-8 LANGUAGE=en_US:en LC_ALL=en_US.UTF-8	0 B
6 RUN /bin/sh -c set -eux;	7.98 MB
7 ENV JAVA_VERSION=jdk-21.0.4+7	0 B
8 RUN /bin/sh -c set -eux;	51.2 MB
9 RUN /bin/sh -c set -eux;	139 B
10 COPY entrypoint.sh /__cacert_entrypoint.sh # buildkit	1.4 KB
11 ENTRYPOINT ["/__cacert_entrypoint.sh"]	0 B
12 EXPOSE map[9092/tcp:{}]	0 B
13 USER root	0 B
14 ARG kafka_url=https://home.apache.org/~jlprat/kafka-3.8.0-rc3/kafka_2.13-3.8.0.tgz	0 B
15 ARG build_date=2024-07-23	0 B

Docker Hub Image Kafka

O que é docker?



Imagen

É o empacotamento de uma aplicação (código) com todas as dependências necessárias para a sua execução.



Registry

Um repositório de imagens que é utilizado como consulta de qual imagem utilizar para a construção de um container.



Container

Um contêiner é uma instância em execução de uma imagem. Ele é uma camada de abstração que permite que a aplicação seja executada de forma isolada do sistema operacional host.

O que é Docker?



Docker é uma plataforma de software que facilita a criação, implantação e gerenciamento de aplicativos em contêineres.

**Como usar
essa
ferramenta?**



**Mãos a
obra!**



Criando uma imagem

```
1 package main
2
3 import (
4     "net/http"
5
6     "github.com/gin-gonic/gin"
7 )
8
9 type User struct {
10    Name string `json:"name"`
11 }
12
13 func main() {
14    user := User{Name: "Mário"}
15
16    r := gin.Default()
17    r.GET("/users", func(c *gin.Context) {
18        c.JSON(http.StatusOK, gin.H{
19            "user": user,
20        })
21    })
22    r.Run()
23 }
```

Usarei um código escrito em Go que expõe uma rota na porta 8080. Ao executar esta API e acessar

`http://localhost:8080/users`

Deverá ser retornado o seguinte JSON:

`{"user":{"name":"Mário"}}`

Ao consultar nosso registry local através do comando:

```
$ docker image ls -a
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

docker image ls -a

é possível ver que não temos nenhuma imagem local.

Dockerfile

```
dockerfile > ...
1  FROM golang:1.23-alpine AS builder
2  WORKDIR /app
3  COPY go.mod go.sum ./
4  RUN go mod download
5  COPY . .
6  RUN go build -o myapp
7
8  FROM alpine:latest
9  WORKDIR /root/
10 COPY --from=builder /app/myapp .
11 EXPOSE 8080
12 CMD ["./myapp"]
```

Um Dockerfile é um script de texto que contém uma série de instruções para automatizar a criação de uma imagem Docker. Essas instruções definem como a imagem deve ser construída, incluindo a base da imagem, a instalação de dependências, a cópia de arquivos, a configuração de variáveis de ambiente e a especificação do comando que será executado quando um contêiner for iniciado a partir dessa imagem.

Anatomia do dockerfile

```
⌚ dockerfile > ...
1  FROM golang:1.23-alpine AS builder
2
3  WORKDIR /app
4
5  COPY go.mod go.sum .
6
7  RUN go mod download
8
9  COPY . .
10
11 RUN go build -o myapp
12
13
14 FROM alpine:latest
15
16 WORKDIR /root/
17
18 COPY --from=builder /app/myapp .
19
20 EXPOSE 8080
21
22 CMD ["./myapp"]
```

Como se pode ver na imagem, existem algumas keywords no dockerfile como FROM, RUN, COPY...

Iremos ver cada uma delas e suas funções.

Anatomia do dockerfile - FROM

O comando FROM no Dockerfile é utilizado para especificar a imagem base a partir da qual o contêiner será construído. Ele pode ser usado várias vezes em um Dockerfile para criar diferentes estágios de construção.

No caso da imagem ao lado, na linha 1 especifica que a imagem base para este estágio de construção é `golang:1.23-alpine`. Esta imagem contém o ambiente necessário para compilar um projeto Go. O alias `builder` é dado a este estágio para que ele possa ser referenciado posteriormente.

E na linha 14, especifica que a imagem base para o estágio final é `alpine:latest`. Esta é uma imagem mínima do Alpine Linux que será usada para criar uma imagem leve contendo apenas o binário compilado.

```
dockerfile > ...
1  FROM golang:1.23-alpine AS builder
2
3  WORKDIR /app
4
5  COPY go.mod go.sum ./
6
7  RUN go mod download
8
9  COPY . .
10
11 RUN go build -o myapp
12
13
14 FROM alpine:latest
15
16 WORKDIR /root/
17
18 COPY --from=builder /app/myapp .
19
20 EXPOSE 8080
21
22 CMD ["../myapp"]
```

Anatomia do dockerfile - WORKDIR

O comando WORKDIR no Dockerfile é utilizado para definir o diretório de trabalho dentro do contêiner. Todos os comandos subsequentes (COPY, RUN, etc.) serão executados a partir deste diretório, a menos que um diretório diferente seja especificado.

No caso da imagem ao lado, na linha 3 define /app como o diretório de trabalho para o estágio de construção chamado builder. Todos os comandos subsequentes neste estágio serão executados a partir deste diretório.

E na linha 16, define /root/ como o diretório de trabalho para o estágio final. Todos os comandos subsequentes neste estágio serão executados a partir deste diretório.

```
dockerfile > ...
1 FROM golang:1.23-alpine AS builder
2
3 WORKDIR /app
4
5 COPY go.mod go.sum ./
6
7 RUN go mod download
8
9 COPY . .
10
11 RUN go build -o myapp
12
13
14 FROM alpine:latest
15
16 WORKDIR /root/
17
18 COPY --from=builder /app/myapp .
19
20 EXPOSE 8080
21
22 CMD ["../myapp"]
```

Anatomia do dockerfile - COPY

O comando COPY no Dockerfile é utilizado para copiar arquivos e diretórios do sistema de arquivos do host para o sistema de arquivos do contêiner.

No caso da imagem ao lado, na linha 5 Copia os arquivos go.mod e go.sum do diretório atual no host(máquina local) para o diretório /app no contêiner (definido anteriormente pelo comando WORKDIR /app).

Na linha 9, Copia todos os arquivos e diretórios do diretório atual no host para o diretório /app no contêiner.

E na linha 18, copia o arquivo myapp do diretório /app no estágio de construção chamado builder para o diretório /root no estágio final do contêiner.

```
git dockerfile > ...
1 FROM golang:1.23-alpine AS builder
2
3 WORKDIR /app
4
5 COPY go.mod go.sum ./
6
7 RUN go mod download
8
9 COPY . .
10
11 RUN go build -o myapp
12
13
14 FROM alpine:latest
15
16 WORKDIR /root/
17
18 COPY --from=builder /app/myapp .
19
20 EXPOSE 8080
21
22 CMD ["../myapp"]
```

Anatomia do dockerfile - EXPOSE

O comando EXPOSE no Dockerfile é utilizado para informar ao Docker que o contêiner escutará em uma porta específica durante a execução.

No nosso dockerfile indica que o contêiner irá escutar na porta 8080. Isso é útil para documentar e configurar quais portas o contêiner usará para se comunicar com o mundo exterior. No entanto, o comando EXPOSE por si só não publica a porta no host; ele apenas informa ao Docker sobre a porta que será usada. Para realmente publicar a porta e torná-la acessível a partir do host, você precisa usar a opção -p ou --publish ao executar o contêiner com o comando docker run, ou configurar a publicação de portas no docker-compose.yml.

Exemplo de uso junto com o comando docker run:

```
docker run -p 8080:8080 <nome-do-seu-contêiner>
```

Mais a frente iremos ver como expor no docker-compose.

```
git clone https://github.com/luizotavio/docker-expose.git
cd docker-expose
cat Dockerfile
```

```
FROM golang:1.23-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o myapp
FROM alpine:latest
WORKDIR /root/
COPY --from=builder /app/myapp .
EXPOSE 8080
CMD ["../myapp"]
```

Anatomia do dockerfile - RUN

O comando RUN no Dockerfile é utilizado para executar comandos no contêiner durante o processo de construção da imagem. Cada comando RUN cria uma nova camada na imagem do Docker.

Novamente olhando para o nosso dockerfile, temos a utilização do comando RUN em duas linhas, 7 e 11, respectivamente uma executa o comando para realizar o download de todas as dependências necessárias para a execução da imagem e a outra compila o código fonte da aplicação Go e gera um binário executável chamado myapp.

```
git dockerfile > ...
1 FROM golang:1.23-alpine AS builder
2
3 WORKDIR /app
4
5 COPY go.mod go.sum ./
6
7 RUN go mod download
8
9 COPY . .
10
11 RUN go build -o myapp
12
13
14 FROM alpine:latest
15
16 WORKDIR /root/
17
18 COPY --from=builder /app/myapp .
19
20 EXPOSE 8080
21
22 CMD ["../myapp"]
```

Anatomia do dockerfile - CMD

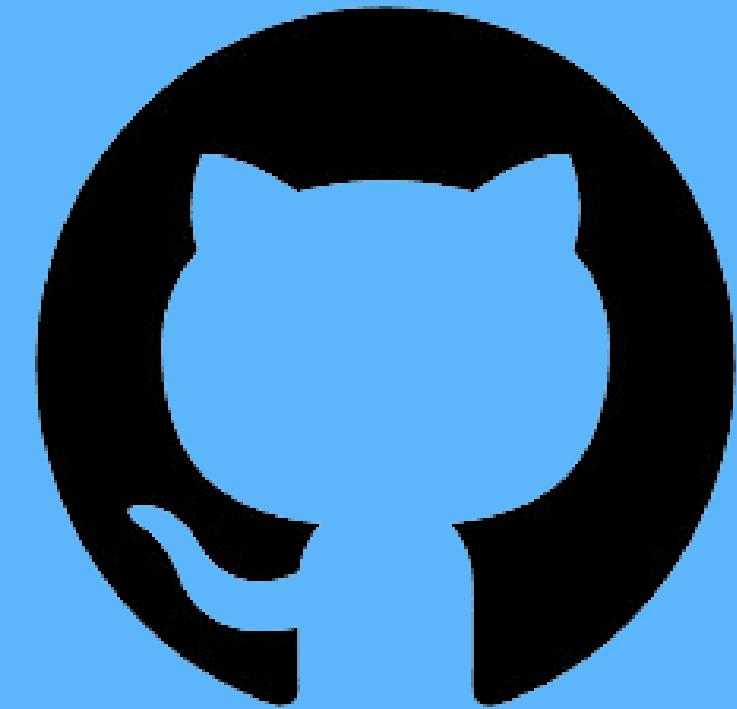
O comando CMD no Dockerfile é utilizado para especificar o comando padrão que será executado quando um contêiner iniciado a partir da imagem for executado. Diferente do comando RUN, que é executado durante a construção da imagem, o CMD é executado quando o contêiner está em execução.

Na linha 22, o comando CMD está configurado para executar o binário myapp. Isso significa que, quando um contêiner é iniciado a partir dessa imagem, ele executará o comando ./myapp por padrão. Se um comando diferente for especificado na linha de comando docker run, ele substituirá o comando CMD.

```
git dockerfile > ...
1 FROM golang:1.23-alpine AS builder
2
3 WORKDIR /app
4
5 COPY go.mod go.sum ./
6
7 RUN go mod download
8
9 COPY . .
10
11 RUN go build -o myapp
12
13
14 FROM alpine:latest
15
16 WORKDIR /root/
17
18 COPY --from=builder /app/myapp .
19
20 EXPOSE 8080
21
22 CMD ["../myapp"]
```

O arquivo dockerfile que
criamos, junto com o código,
está disponível no GITHUB
através do link abaixo:

<https://github.com/andreleao1/fiap-docker/tree/feature/criating-dockerfile>



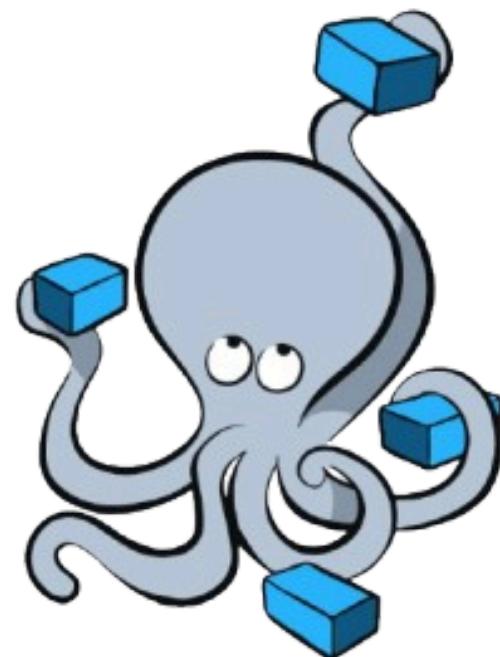
Orquestrando Imagens com Docker Compose

Voltemos para analogias...



Vamos imaginar uma orquestra onde temos diversos instrumentos, onde cada um possui sua especificidade, afinação e etc... Agora imagine essa orquestra cada um tocando notas aleatórias simultaneamente... Acredito que nossa experiência como ouvintes não seria uma das melhores. Para organizar e apontar qual instrumento vai tocar o que e quando, temos a presença o regente.

Definindo Docker Compose



Agora vamos imaginar o cenário onde temos a necessidade de executar 10 contêineres simultaneamente. Para você acha que seria performático executar 10 vezes o comando docker run ? E para definir a especificidade de cada imagem?

Temos de concordar que seria uma trabalheira grande para fazer isso.

Então para nos ajudar com esta demanda podemos contar com o docker compose podemos definir e gerenciar multi-containers Docker applications. Para isso é utilizado um arquivo YAML para configurar os serviços da sua aplicação, e com um único comando, pode criar e iniciar todos os serviços a partir dessa configuração.

Vamos voltar para nossa aplicação Go e adicionar a necessidade de nos conectarmos a um banco de dados para consultar os usuários a partir de uma tabela chamada users.

Vou utilizar o comando docker image ls -a para verificar que nossa imagem continua no nosso registry local e como podemos ver na imagem, até aqui está tudo ok.

```
$ docker image ls -a
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
poc_docker      latest    8bc451ac05a3  2 hours ago  19.1MB
```

Anatomia do Docker Compose file

```
👉 docker-compose.yaml  
1  version: '3.8'  
2  services:  
3    go-app-users:  
4      image: poc_docker  
5      ports:  
6        - "8080:8080"  
7      depends_on:  
8        - postgres  
9      environment:  
10        - DB_HOST=postgres  
11        - DB_USER=postgres  
12        - DB_PASSWORD=postgres  
13        - DB_NAME=postgres  
14        - DB_PORT=5432  
15  
16    postgres:  
17      image: postgres:15  
18      environment:  
19        POSTGRES_USER: postgres  
20        POSTGRES_PASSWORD: postgres  
21        POSTGRES_DB: postgres  
22      ports:  
23        - "5432:5432"
```

O Docker Compose utiliza a estrutura YAML (YAML Ain't Markup Language), um formato de serialização de dados legível por humanos, comumente usado para arquivos de configuração.

No contexto do Docker, o arquivo docker-compose.yaml é utilizado para definir e executar aplicativos multi-contêiner através do uso de algumas keywords.

Para executar um docker compose utilizamos os comandos docker compose up, que pode ser seguido da tag -d para que o terminar possa ser liberado. E para encerrar a execução dos containers é utilizado o comando docker compose down.

Na imagem ao lado, podemos ver um exemplo de um arquivo YAML que define um contêiner para uma aplicação(nossa aplicação GO) e outro contêiner para um banco de dados PostgreSQL.

Anatomia do dockerfile - Version

Especifica a versão do formato do arquivo de composição (docker-compose.yaml), determinando os recursos e sintaxes disponíveis.

Versões possíveis:

- '1': Versão inicial, limitada, não recomendada para novos projetos.
- '2': (Docker Compose 1.6.0) Suporta depends_on, networks, volumes.
- '2.1': (Docker Compose 1.8.0) Adiciona suporte para secrets.
- '3': (Docker Compose 1.10.0) Focada em clusters Docker Swarm.
- '3.1': (Docker Compose 1.13.0) Suporte para configs.
- '3.2': (Docker Compose 1.16.0) Suporte para deploy, resources.
- '3.3': (Docker Compose 1.18.0) Suporte para init.
- '3.4': (Docker Compose 1.21.0) Suporte para target em build.
- '3.5': (Docker Compose 1.22.0) Suporte para link_local_ips.
- '3.6': (Docker Compose 1.23.0) Suporte para credential_spec.
- '3.7': (Docker Compose 1.25.0) Suporte para runtime.
- '3.8': (Docker Compose 1.27.0) Suporte para profiles.

```
docker-compose.yaml
1 version: '3.8'
2 services:
3   go-app-users:
4     image: poc_docker
5     ports:
6       - "8080:8080"
7     depends_on:
8       - postgres
9     environment:
10      - DB_HOST=postgres
11      - DB_USER=postgres
12      - DB_PASSWORD=postgres
13      - DB_NAME=postgres
14      - DB_PORT=5432
15
16   postgres:
17     image: postgres:15
18     environment:
19       POSTGRES_USER: postgres
20       POSTGRES_PASSWORD: postgres
21       POSTGRES_DB: postgres
22     ports:
23       - "5432:5432"
```

Anatomia do dockerfile - Services

A keyword services no Docker Compose é usada para definir os contêineres que compõem a sua aplicação. Cada serviço é um contêiner que será gerenciado pelo Docker Compose. Dentro de cada serviço, você pode especificar várias configurações, como a imagem do contêiner, volumes, portas, variáveis de ambiente, entre outras. Irei entrar no detalhes das propriedades utilizadas nesta aula e conforme formos avançando vou introduzindo novas configurações... Keep Calm!

```
docker-compose.yaml
1 version: '3.8'
2 services:
3   go-app-users:
4     image: poc_docker
5     ports:
6       - "8080:8080"
7     depends_on:
8       - postgres
9     environment:
10      - DB_HOST=postgres
11      - DB_USER=postgres
12      - DB_PASSWORD=postgres
13      - DB_NAME=postgres
14      - DB_PORT=5432
15
16   postgres:
17     image: postgres:15
18     environment:
19       POSTGRES_USER: postgres
20       POSTGRES_PASSWORD: postgres
21       POSTGRES_DB: postgres
22     ports:
23       - "5432:5432"
```

Anatomia do dockerfile - Services - image

Especifica a imagem do Docker a ser usada para o contêiner. No nosso caso, estamos definindo duas imagens, poc_docker que é a imagem que definimos através do dockerfile e a imagem do postgres 15.

```
docker-compose.yaml
1  version: '3.8'
2  services:
3    go-app-users:
4      image: poc_docker
5      ports:
6        - "8080:8080"
7      depends_on:
8        - postgres
9      environment:
10        - DB_HOST=postgres
11        - DB_USER=postgres
12        - DB_PASSWORD=postgres
13        - DB_NAME=postgres
14        - DB_PORT=5432
15
16    postgres:
17      image: postgres:15
18      environment:
19        POSTGRES_USER: postgres
20        POSTGRES_PASSWORD: postgres
21        POSTGRES_DB: postgres
22      ports:
23        - "5432:5432"
```

Anatomia do dockerfile - Services - ports

Lembra que foi mencionado que só definir a porta na construção da imagem não seria suficiente para expor a porta para o host? E que precisaríamos definir no comando docker run passando a flag -p ou definir no docker compose?

Então aqui estamos, usamos a configuração ports para definir qual porta da imagem é exposta e qual porta do container será exposta. No docker compose de exemplo, estamos expondo a porta 8080 para a aplicação Go e a porta 5432 para o banco de dados.

```
docker-compose.yaml
1 version: '3.8'
2 services:
3   go-app-users:
4     image: poc_docker
5     ports:
6       - "8080:8080"
7     depends_on:
8       - postgres
9     environment:
10      - DB_HOST=postgres
11      - DB_USER=postgres
12      - DB_PASSWORD=postgres
13      - DB_NAME=postgres
14      - DB_PORT=5432
15
16   postgres:
17     image: postgres:15
18     environment:
19       POSTGRES_USER: postgres
20       POSTGRES_PASSWORD: postgres
21       POSTGRES_DB: postgres
22     ports:
23       - "5432:5432"
```

Anatomia do dockerfile - Services - environment

Define variáveis de ambiente para o contêiner. No nosso caso precisamos fornecer para a nossa imagem quais são as variáveis de ambiente para acessar o banco e para a nossa imagem postgres, precisamos definir senha, nome de usuário e banco de dados.

```
docker-compose.yaml
1 version: '3.8'
2 services:
3   go-app-users:
4     image: poc_docker
5     ports:
6       - "8080:8080"
7     depends_on:
8       - postgres
9     environment:
10      - DB_HOST=postgres
11      - DB_USER=postgres
12      - DB_PASSWORD=postgres
13      - DB_NAME=postgres
14      - DB_PORT=5432
15
16   postgres:
17     image: postgres:15
18     environment:
19       POSTGRES_USER: postgres
20       POSTGRES_PASSWORD: postgres
21       POSTGRES_DB: postgres
22     ports:
23       - "5432:5432"
```

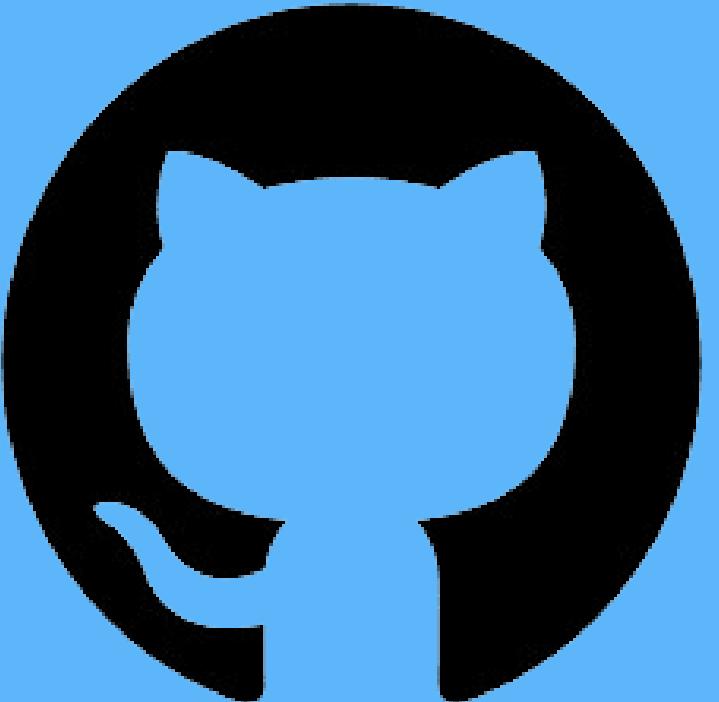
Anatomia do dockerfile - Services - depends_on

Define dependências entre serviços. No nosso caso, o container da aplicação só irá ser executado quando o container postgres estiver rodando.

```
docker-compose.yaml
1  version: '3.8'
2  services:
3    go-app-users:
4      image: poc_docker
5      ports:
6        - "8080:8080"
7      depends_on:
8        - postgres
9      environment:
10        - DB_HOST=postgres
11        - DB_USER=postgres
12        - DB_PASSWORD=postgres
13        - DB_NAME=postgres
14        - DB_PORT=5432
15
16    postgres:
17      image: postgres:15
18      environment:
19        POSTGRES_USER: postgres
20        POSTGRES_PASSWORD: postgres
21        POSTGRES_DB: postgres
22      ports:
23        - "5432:5432"
```

O arquivo docker compose que criamos, junto com o código, está disponível no GITHUB através do link abaixo:

<https://github.com/andreleao1/fiap-docker/tree/feature/using-dockercompose>

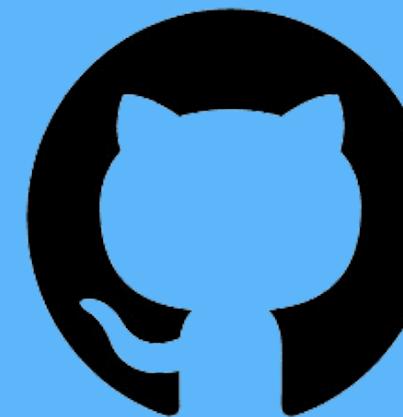


Obrigado!



Dúvidas

andregustavols4@gmail.com



Código e Material

[GITHUB](#)