Docker

Containers, Imagens e Cluster

SemComp 2017 - ICMC / USP São Carlos

Quem sou eu

Formado em Eng Computação pela USP (ICMC / EESC) em 2008. Trabalhei em sistemas tolerantes à falha de alto volume.

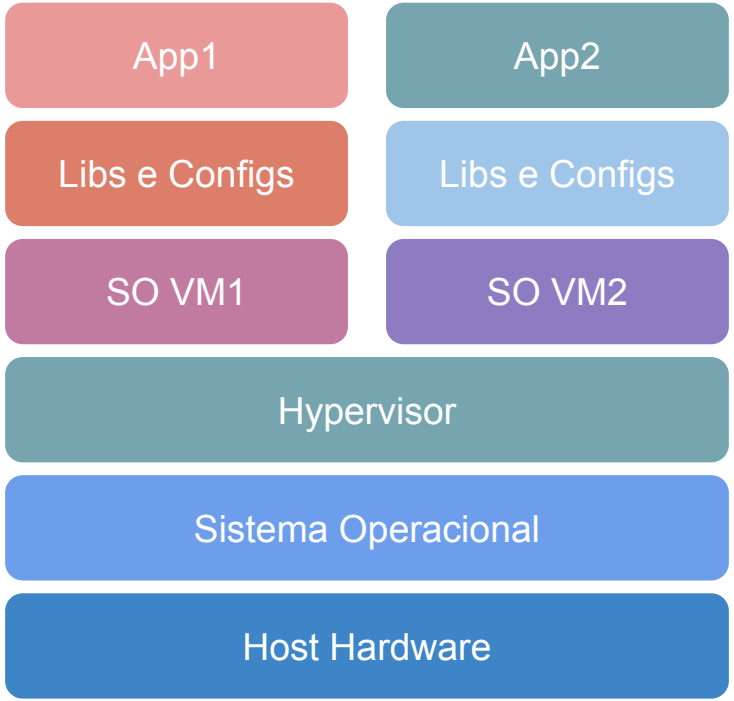
Atuei como desenvolvedor Back End, Mobile, Líder Técnico, Gerente de Projetos e de Contas.

Em 2016 entrei para a startup CasaeCafe.com onde atuo como Ops, Arquiteto de Software e CTO.

Virtualização

Como funciona uma Máquina Virtual?

- Quais as vantagens de utilizar uma VM ao invés de uma máquina física?
- Em quais situações a VM é melhor?
- Como uma VM provê essas vantagens?



Camadas de um sistema virtualizado com VM

Problemas

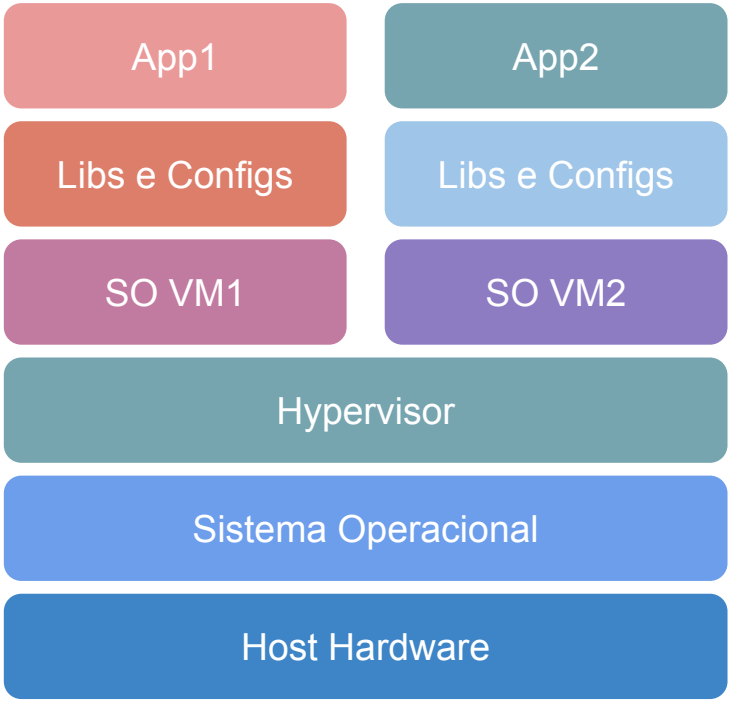
- Overhead de **processamento** e consumo de **memória**
- Tempo de subida de alguns **minutos** (VM)
- **Imagens grandes**, pois possuem toda a instalação do sistema operacional
- Baixo **reuso**
- Execução de **diversos processos** simultâneos, dificultando debug
    - Exemplo: Máquina Virtual instável
- Dificuldade de orquestrar muitas VMs

LinuX Containers (LXC)

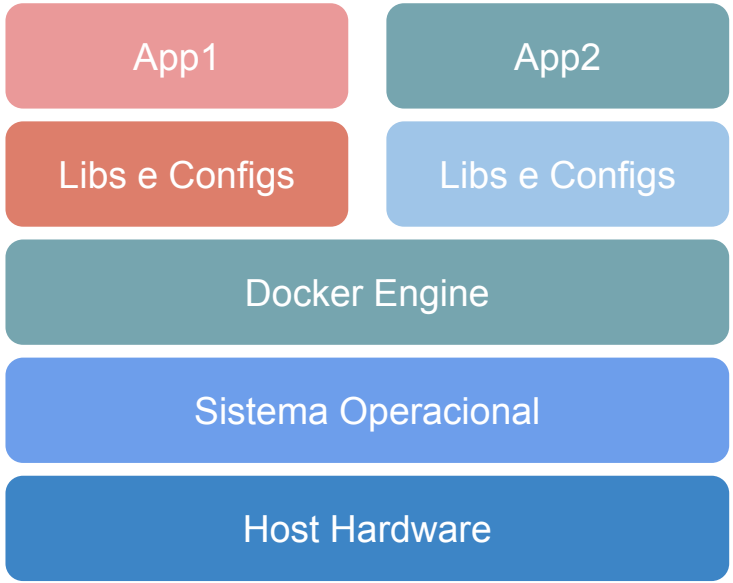
- Compartilhamento de Kernel
- Isolamento de Árvore de Processos
- Isolamento de pastas ~ chroot
- Isolamento de consumo de consumo de recursos ~ crgroup
- Isolamento de rede

Docker

- Facilitador para utilização de Linux Containers
- Toolset para gerenciamento de containers
- Grande ecossistemas



Máquina Virtual



Docker

Container

- Processo pai com sua árvore de processos.
    - Evitar múltiplos processos

      Ex: BD, BackEnd, Jobs, Servidor de Aplicação
- Sistema de arquivos isolado (chroot)
- Encapsulamento das dependências mínimas
- Ambiente leve
- Garante imutábilidade entre diversos ambientes
- Criado a partir de uma imagem

Imagem

- Sistema de arquivo base para o container
  - Contém todos os arquivos necessários para iniciar containers
  - O container é o processo em execução neste sistema de arquivos
- Criação a partir de build do Dockerfile ou commit de um container
  - Boa prática: Usar build
- Armazenamento em repositórios de um Registry
  - Versionamento de imagens a partir de tags
  - Variações de imagens com diferentes SOs
    - Apline: distribuição bem leve de linux

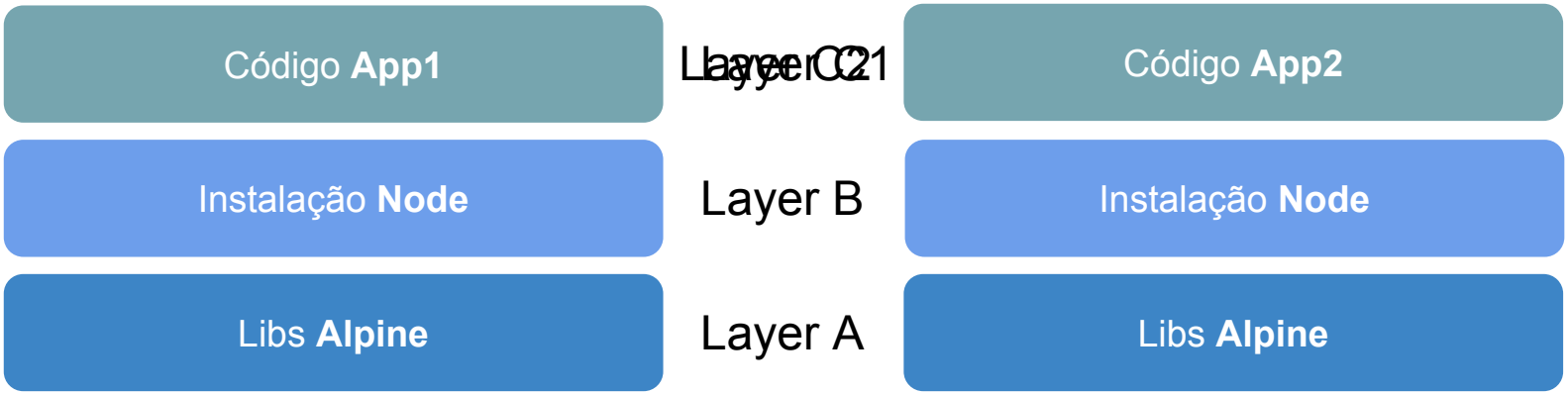
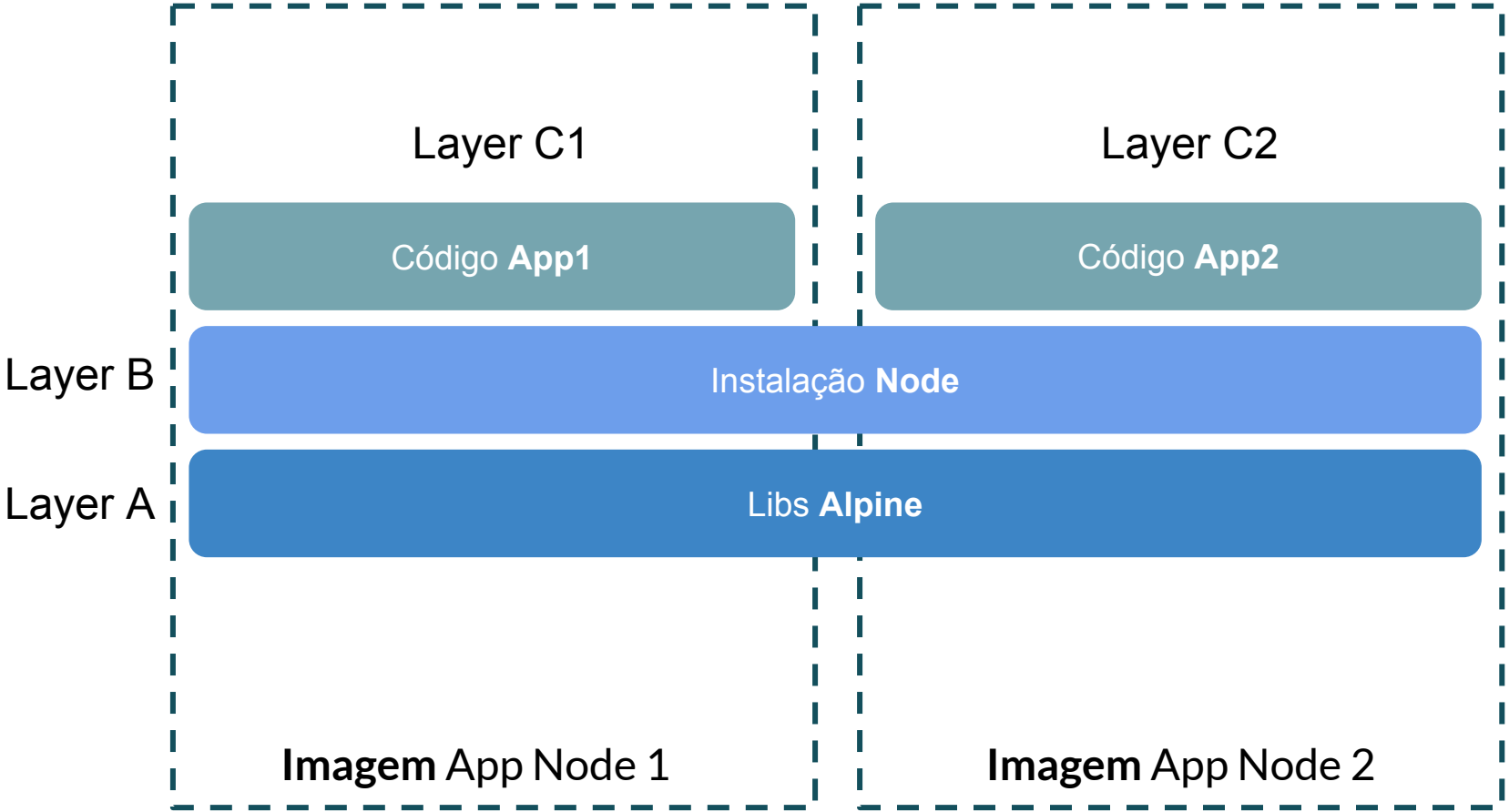


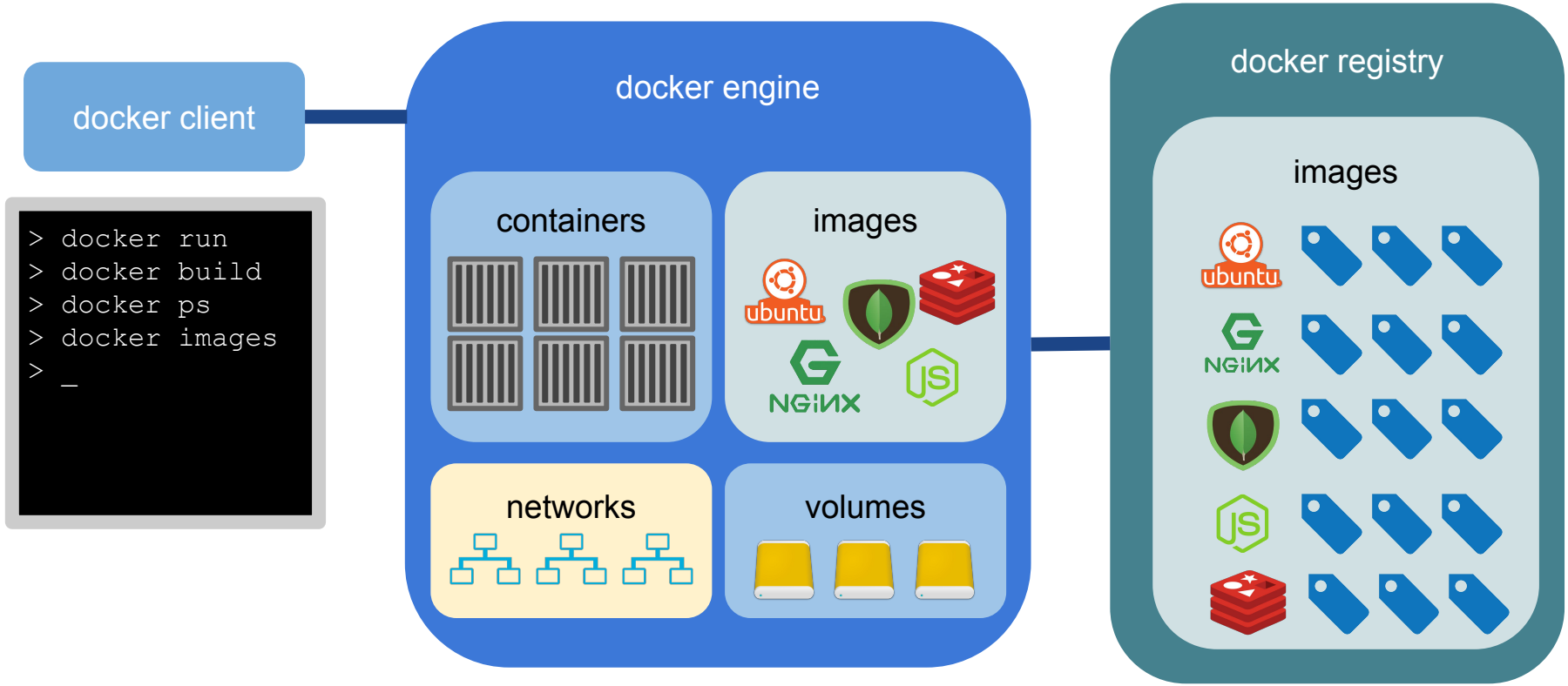
Imagem App Node 1

Imagem App Node 2


Layer C1
Layer C2
Código App1
Código App2
Layer B
Instalação Node
Layer A
Libs Alpine
Imagem App Node 1
Imagem App Node 2

Imagens e Containers

- Imagem é o modelo com o qual um container é criado
  - POO: Classe (modelo) e Objeto (instância do modelo, com estado)
  - Receita e Bolo
- Todo container é criado a partir de uma imagem
- Múltiplos contaneirs podem ser baseados na mesma imagem
- Containers usam as layers de uma imagem com uma layer de leitura e escrita no topo
- Sistema de arquivo mais comum é o AUFS (Another Union File System)
  - Copy on Write


docker client
> docker run
> docker build
> docker ps
> docker images
> _
docker engine
containers
images
networks
volumes
docker registry
images

Instalação Docker

Linux - Deamon

Windows - API Kernel + Deamon

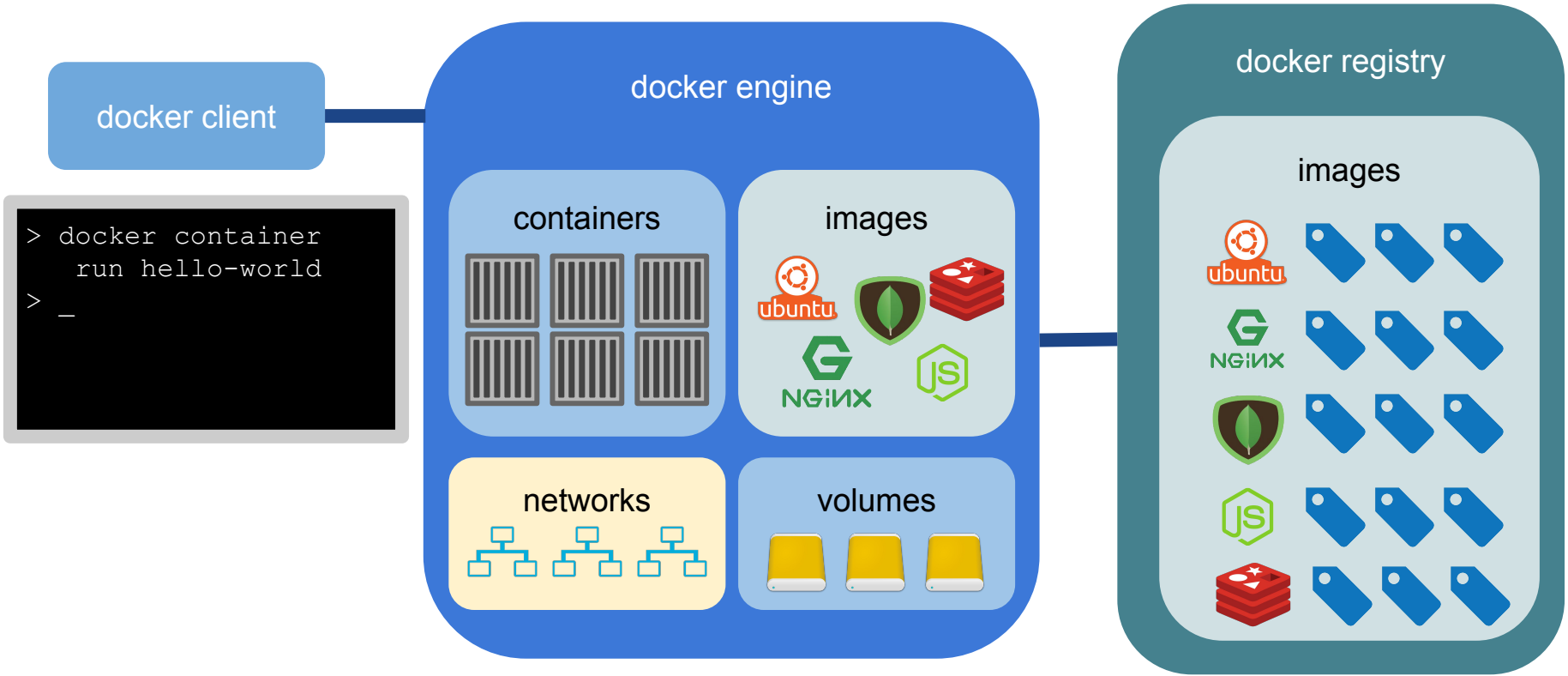
Mac OS - VM Linux

Começando a Brincadeira

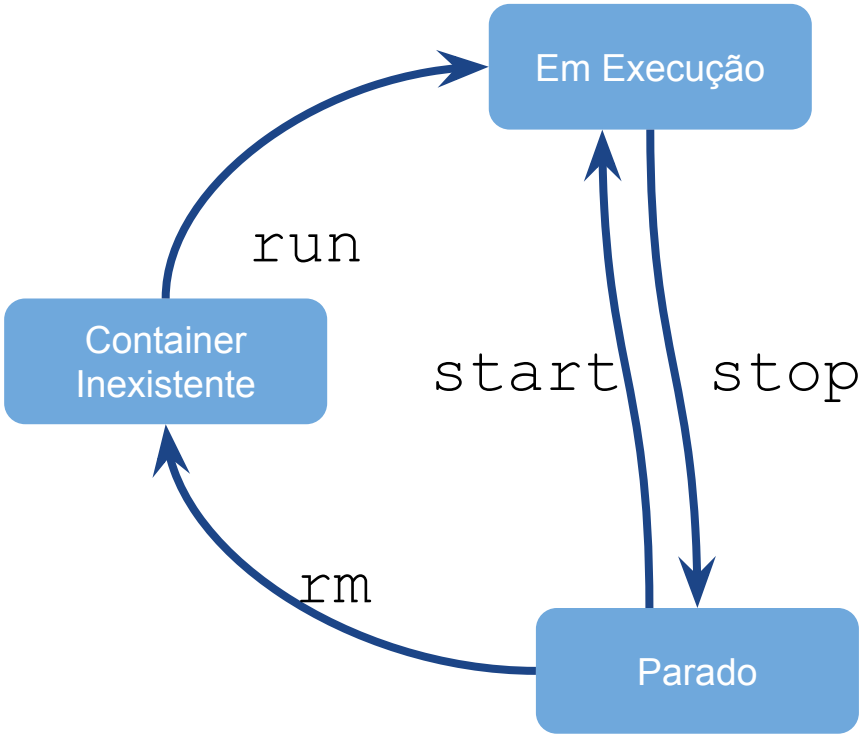
Olá, Oceano!!!

```
> docker container run hello-world
```

O que aconteceu aqui?


docker client
> docker container
run hello-world
> _
docker engine
containers
images
networks
volumes
docker registry
images

Docker Container Lifecycle



Modos de Execução

-d: execução em background (detached)
-it: iterativo
--rm: remover ao parar

Exemplos

bash no *Ubuntu*

Executar *NGINX*

Rodar o *MongoDB*

```
> docker container run -it ubuntu
bash


> docker container run -p 8080:80 -d
nginx


> docker container run --name c_mongo
-d -v $PWD:/data/db -d mongo
> docker exec -it c_mongo mongo
```

Aprendizado

- Para acessar portas de dentro do container, precisamos mapeá-las no HOST
- O container deve ser stateless, pois a qualquer momento pode ser reiniciado
- Os arquivos do container devem ser mapeados para um volume no HOST
- Podemos usar comandos exec para "entrar" no container

```
> docker container run
```

```
> docker container exec
```

Lego com Containers

Juntando tudo! :)

Vamos fazer um serviço **REST** em **Node.js** que escreve e lê documentos no **MongoDB**

Código no Repositório:

https://github.com/rafapg/semcomp2017

Passo a Passo #1 _MongoDB

Precisamos inicializar o *MongoDB*, que será nosso banco de Dados. Nosso container tem algumas necessidades:

- Deve ser *detached*
- Precisamos mapear o *volume* para não perder os dados
- É conveniente dar um *nome* para o container

```
> docker run --name c_mongo --rm -d mongo
```

Passo a Passo #2 _Dependências

Agora vamos ao serviço *Node.js*.

- Precisamos baixar o código no repositório

```
> git clone https://github.com/rafapg/semcomp2017
```

- Agora vamos instalar as dependências do projeto. Para isso precisamos de ter o *npm* instalado... Será mesmo???

```
> docker run --rm  -it -v $PWD:/usr/src/app -w
/usr/src/app node npm install
```

Passo a Passo #3 _Server Rest

Como diria o Waze: *"Tudo pronto? Vamos!"*

Vamos inicializar o nosso servidor. O que precisamos levar em conta?

- O servidor deve ser acessível via alguma porta (p.ex. 8080)
- Mapeamento do diretório do projeto para dentro do container (*volume*)
- Definir a pasta de trabalho que o *Node.js* vai utilizar

```
> docker run --name node_server -v $PWD:/usr/src/app -w
    /usr/src/app -d -p 8080:3000 node npm run start
```

Passo a Passo #4 _Erro de Conexão com DB

Putz... Deu ruim!

MongoError: failed to connect to server [c_mongo:27017] on first connect

- Temos que configurar o IP do Container *c_mongo* no Mongoose

```
> docker container inspect c_mongo -f
"{{".NetworkSettings.Networks.bridge.IPAddress"}}"
```

- Na string de conexão do arquivo *server.js* do projeto

```
mongoose.connect('mongodb://[IP]/SemComp2017');
```

Passo a Passo #5 _Teste dos Serviços

Vamos testar as funcionalidades (com ajuda do Postman).

- Criar alguns Minicursos
- Listar todos os Minicursos
- Apagar um Minicurso
- Obter um Minicurso específico

IP do Mongo *Hard Coded :(*

Em um cenário hipotético que temos diversos serviços acessando o *MongoDb*, imagine o caos que deve ser configurar todos para o IP do container *MongoDb* do ambiente.

Não parece muito prático, Né?!

Vamos linkar os containers

O *docker engine* também faz a função de *service discovery* para os containers. Isso quer dizer que podemos, ao invés de acessar o container pelo IP, chamá-lo diretamente pelo seu nome, contato que realizemos o link

Como isso ficaria?

```
> docker run --name node_server --link c_mongo -v
$PWD:/usr/src/app -w /usr/src/app -d -p 8080:3000 node npm
run start
```

Dicas e Malemolências

Como tirar proveito das funcionalidades de um container:

- Mapeamento de Volumes (pasta host <-> container)
- Exposição de Portas (host <-> container)
- Copia de arquivos para dentro do container e vice-versa
- Utilização de um container por outros containers

Dicas e Malemolências

```
> docker container [comando]
```

`ps` - Listar (`-a` para todos)

`stop` - Parar

`start` - Iniciar

`restart` - Reiniciar

`rm` - Remover (`-f` para forçar)

`logs` - Ver logs (`-f` para tail)

`inspect` - Inspecionar

`exec` - Executar comandos

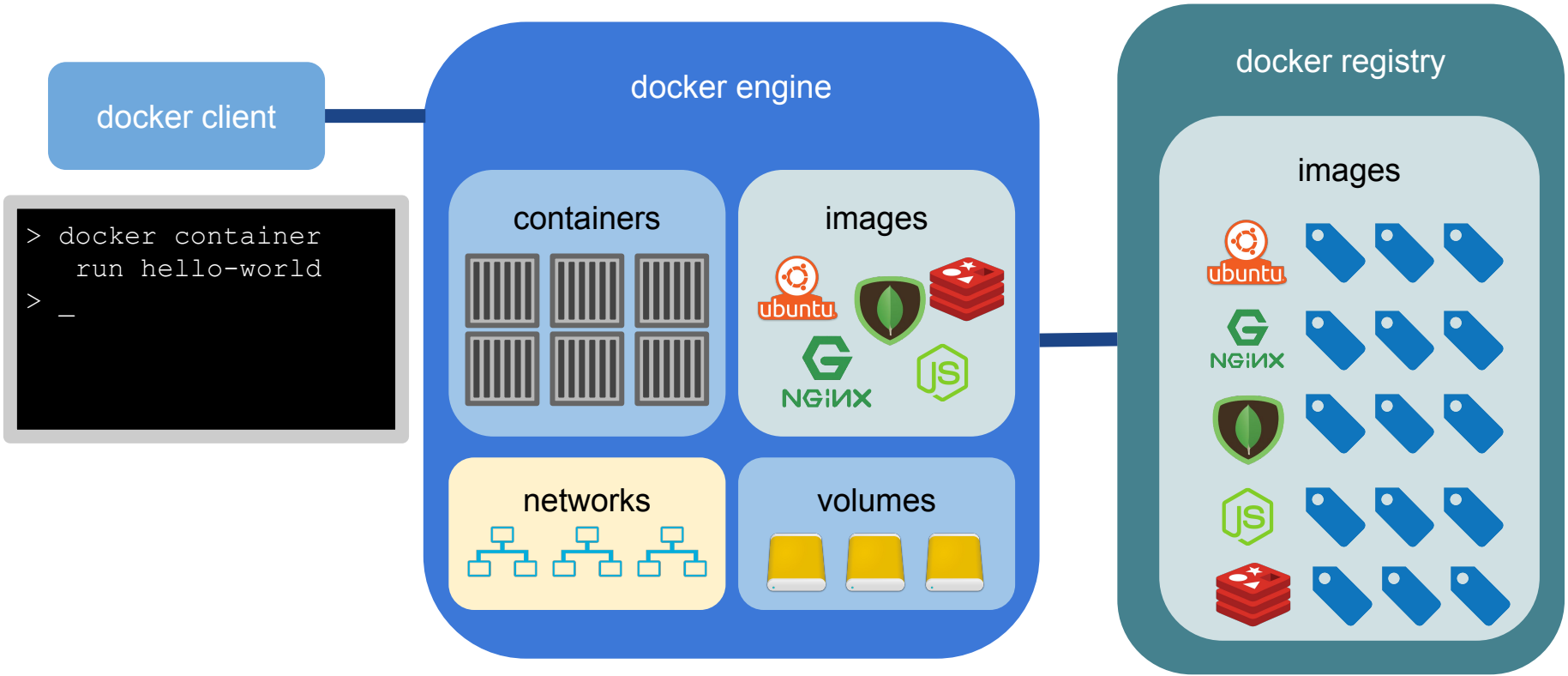
Voltando à teoria

Registry e Repositórios

Registry é um conjunto de repositórios sob uma mesma estrutura ou domínio.

Repositório é uma coleção versões diferentes de uma imagem, separadas por tag

Exemplo: https://hub.docker.com/_/node/


docker client
> docker container
run hello-world
> _
docker engine
containers
images
networks
volumes
docker registry
images

Dicas e Malemolências

```
> docker image [comando]
```

`ls` - Listar

`rm` - Remover

`pull` - Baixar imagem

`push` - Enviar para um repositório

`tag` - Atribui uma tag

`inspect` - Inspecionar

`build` - Gerar imagem a partir do Dockerfile

Cozinhando uma Imagem

Comandos do Dockerfile

```
FROM <imagem>[:<tag>]

RUN <comando_shell>

COPY <path_host> <path_container>

EXPOSE <porta>

ENV <chave> <valor>

WORKDIR <path_container>

ENTRYPOINT <comando_shell>
```

Exemplo NGINX

Criar uma imagem customizada do *NGINX* com o index.html customizado

Imagem Serviço *Node.js*

Vamos agora montar uma imagem com nosso serviço *Node.js*

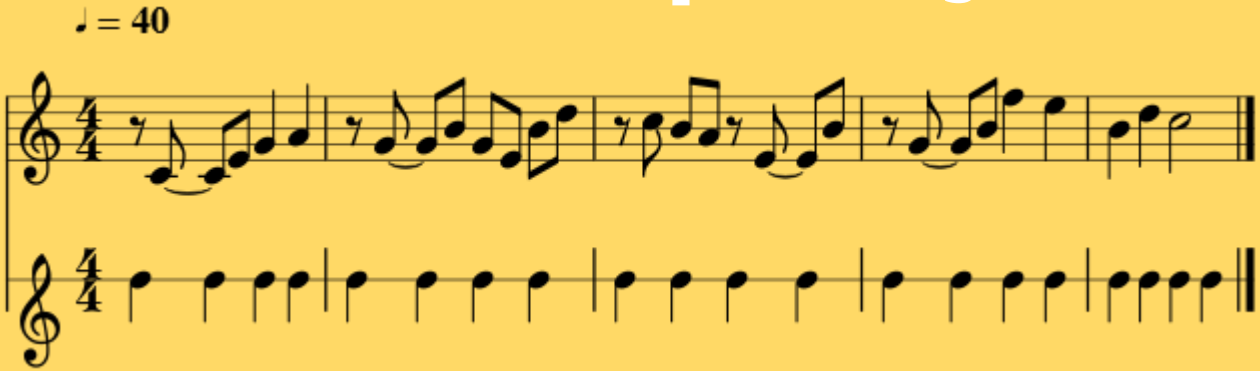
Imagem Serviço *Node.js*

1. Imagem a partir de uma versão estável do node
    a. Versões do alpine deixam a imagem mais leve
2. Copiar o código fonte para o *filesystem*
3. Setar o diretório de trabalho
4. Instalar as dependências
    a. Se possível, garantir que não são usadas dependências velhas
5. Definir o comando de inicialização do container

Layers Serviço *Node.js*

Schaub Global Inc. Container

Composição



Microservices Overview

Arquitetura de Microserviços:

- Separação por Domínios
- Colaboração entre Serviços para prover Funcionalidades
- Baixo acoplamento
- DBs Separados

Docker Compose

Compose é a ferramenta do *Docker* que permite iniciar um ambiente com diversos serviços interdependentes através de um arquivo de configuração com apenas **UM** comando

https://docs.docker.com/compose/compose-file/

Compondo nosso Ambiente

Analisando o comando para executar o *MongoDb*:

```
> docker run --name c_mongo -v $PWD/data:/data/db --rm -d
                                                    mongo
```

- Nome: c_mongo
- Volume: ./data -> /data/db
- Imagem: mongo:latest

Compondo nosso Ambiente

Analisando o comando para executar o *NodeServer*:

```
> docker run --name node_server -d --link c_mongo -p
                8080:3000 node-server:0.0.1
```

- Nome: node_server
- Link: c_mongo
- Depende: Mongo em execução
- Porta: 8080->3000
- Imagem: node-server:0.0.1

docker-compose.yml

```
version: '3'
services:
  c_mongo:
    image: mongo:latest
    volumes:
      - ./node-rest-example/data:/data/db

  node_server:
    image: node-server:0.0.1
    ports:
      - 8080:3000
    links:
      - c_mongo
    depends_on:
      - c_mongo
```

Executando o Ambiente

Para iniciar o ambiente:

```
> docker-compose up -d
```

Para ver os logs:

```
> docker-compose logs -f
```

Para remover o ambiente:

```
> docker-compose down
```

Rede

Redes no Docker

Drivers de Rede:

- Bridge: Default do docker
  - Cria sempre uma nova interface de rede
  - Sub redes separadas: Range de IPs específicos
- None
  - Sem rede
- Host
  - Utiliza a rede do Host, sem uma nova interface de rede
- Overlay
  - Utilizado no modo Swarm

Redes no Docker

As redes no docker são as ligações utilizadas em composições mais complexas

- Segurança e acesso restrito
- Registro de serviços

As alterações na rede podem ser feitas manualmente ou a partir do arquivo de configuração *docker-compose.yml*

A diretiva *link* utilizado anteriormente, pode ser substituído pelo *alias* na rede de cada serviço, se ambos estiverem na mesma rede

docker-compose.yml

```yaml
version: '3'
services:
  mongo_database:
    image: mongo:latest
     networks:
       minicurso:
         aliases:
            - c_mongo
    volumes:
      - ./node-rest-example/data:/data/db
  node_server:
    image: node-server:0.0.1
     networks:
       - minicurso
    ports:
      - 8080:3000
    depends_on:
      - mongo_database
networks:
  minicurso:
    external: false
```

Escalabilidade

Vertical x Horizontal

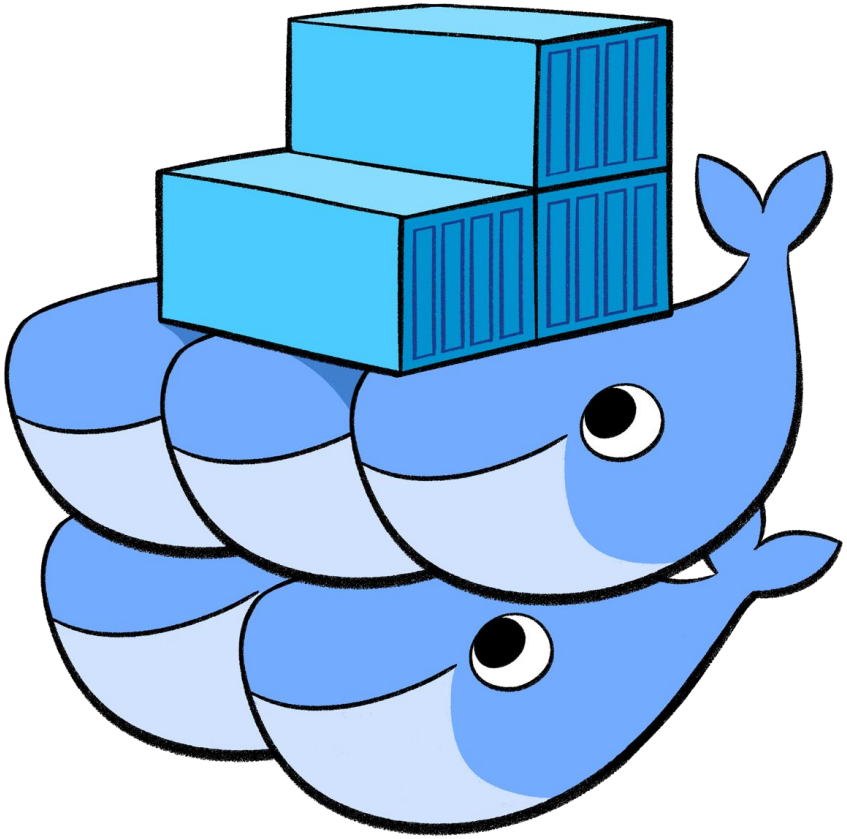
- Escalabilidade Vertical é atingida quando aumentamos o recurso da máquina para aumentar nossa capacidade de processamento
- Escalabilidade Horizontal é atingida quando adicionamos novos nós para aumentar nossa capacidade de processamento

Escalando com apenas uma máquina

É possível escalar horizontalmente com apenas 1 host:

- Diversos containers mesmo serviço, dentro de um pool
- Engine gerencia as requisições

Porém, para utilizar a feature da escalabilidade, é necessário executar em modo Swarm.



Docker Swarm:

- Roteia as requisições entre instâncias do mesmo serviço
- Faz verificação de status
- Mantém o nro de instâncias em execução
- Permite escalar containers em uma ou mais máquinas

Perguntas&
Dúvidas&
Certezas&
Elogios&
Containers

Obrigado!!!

fb.com/rafael.girolineto

@rafakareka

rafapg

rafapg.85@gmail.com