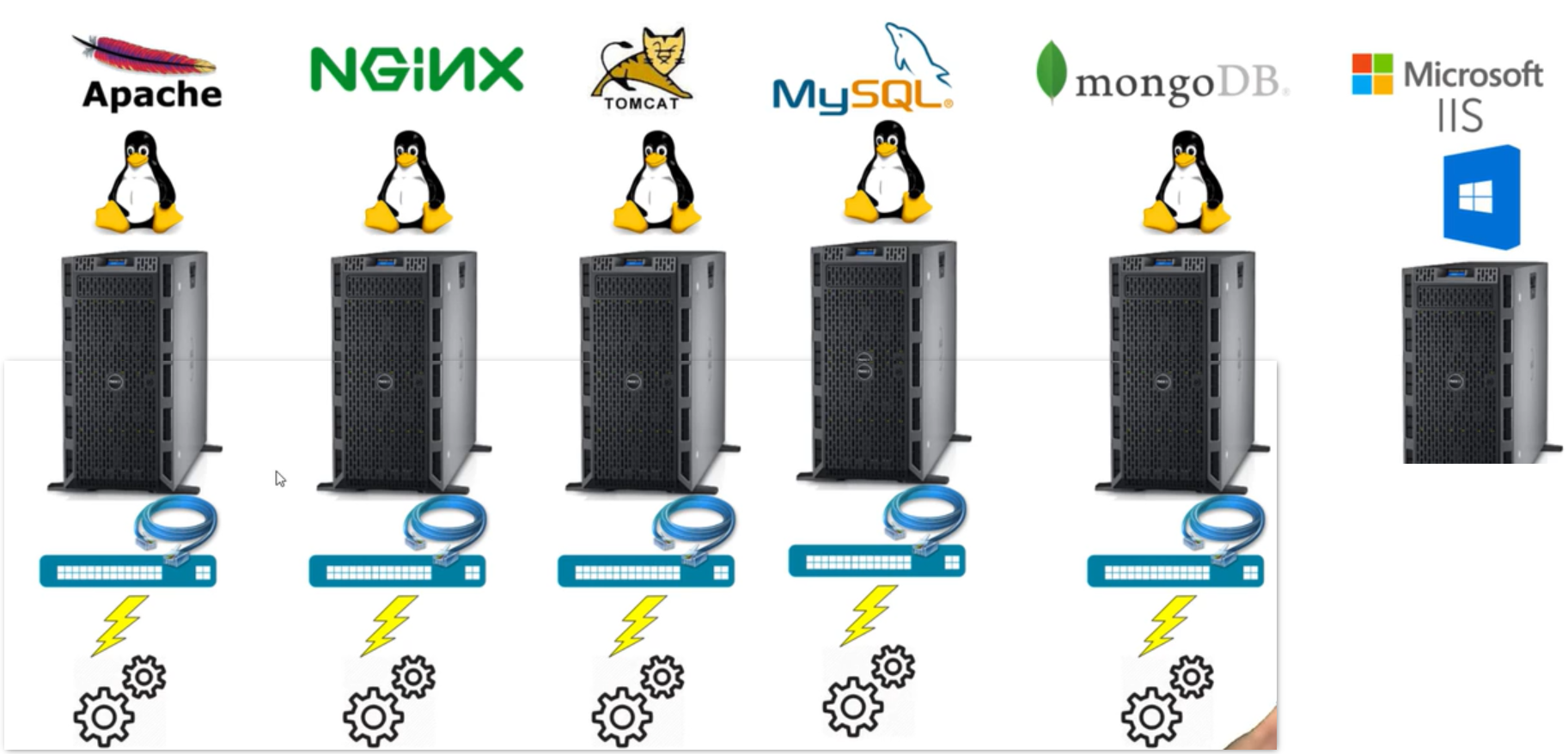
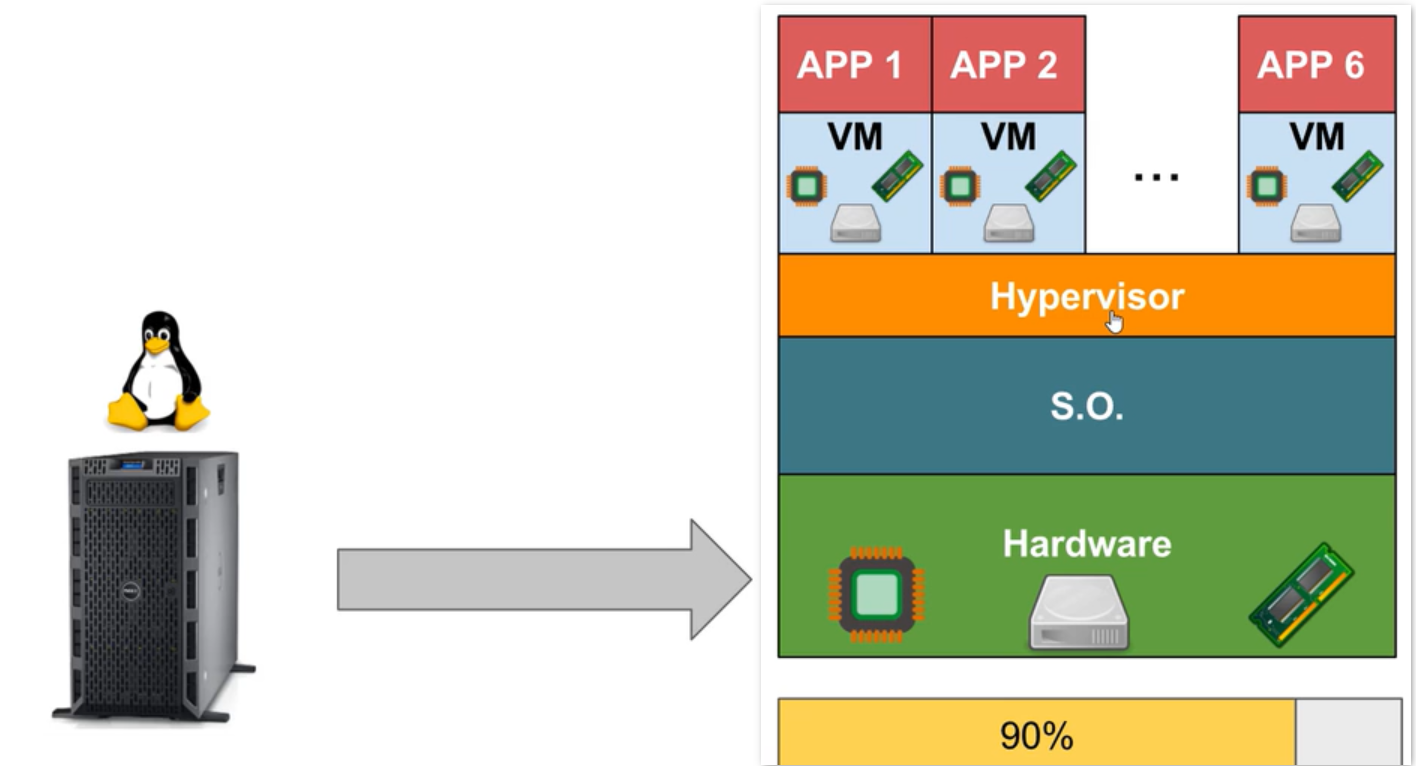
**Docker**

1. **Evolução dos Hosts**

* **UM servidor por aplicação**

****

* **Alto custo** por aplicação criada = Novo servidor
* **Hardware Ocioso:** servidor não usava sua capacidade total com apenas uma aplicação executada
* **Máquinas Virtuais**

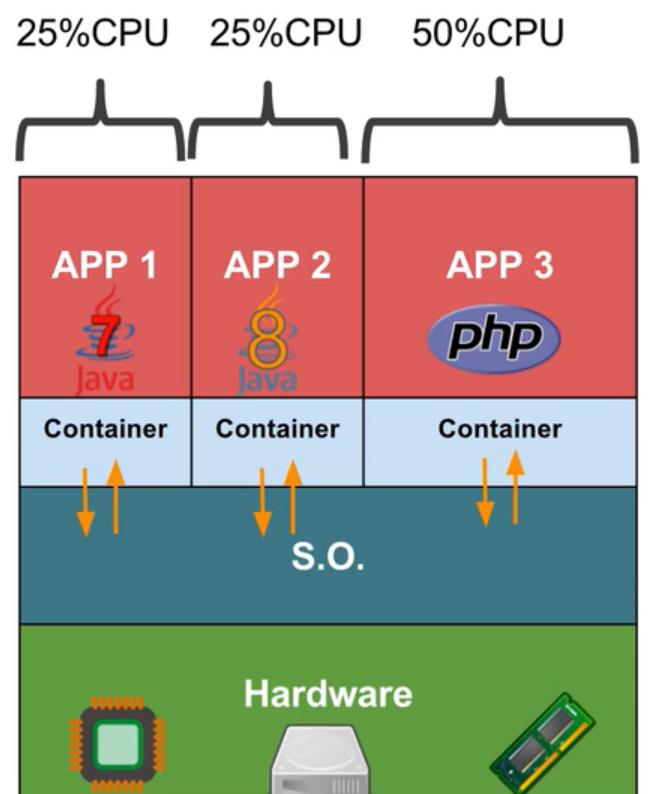
****

* Nova aplicação = Nova VM, não é necessário um servidor/aplicação
* Os Hardwares não ficam mais ociosos: os ambientes das aplicações eram desenvolvidos até a “capacidade máxima”
* **Problema das VM’s**



* Cada VM precisa de seu próprio **Sistema Operacional**
  + Gera custo de **tempo do desenvolvedor**: que precisa estar sempre **Configurando** e **Atualizando** cada sistema operacional de cada máquina virtual
  + Cada máquina virtual necessita de **recursos da máquina física**

1. **Era dos Containers**

****

* Operam no **mesmo Sistema Operacional**, utilizando bem **menos recursos** que a VM
  + A utilização dos recursos entre as APP’s é dividida entre próprios containers (sendo possível limitar o consumo de cada container)
* **PROBLEMA É QUESTIONAR:** Mas por que precisamos dos Containers? Não posso simplesmente instalar as aplicações no nosso próprio Sistema Operacional, uma vez que eles não se conflitam?
* E se 2 aplicativos precisarem utilizar a **mesma porta de rede?**

Seria necessária uma solução pra isolar uma da outra

* E se uma aplicação **consumir toda a CPU** e prejudicar o funcionamento de outra aplicação?
* E se uma aplicação necessitar de uma **versão específica de linguagem/framework**?
* Se uma aplicação cair e precisar **reiniciar o server**, todas as app’s vão cair

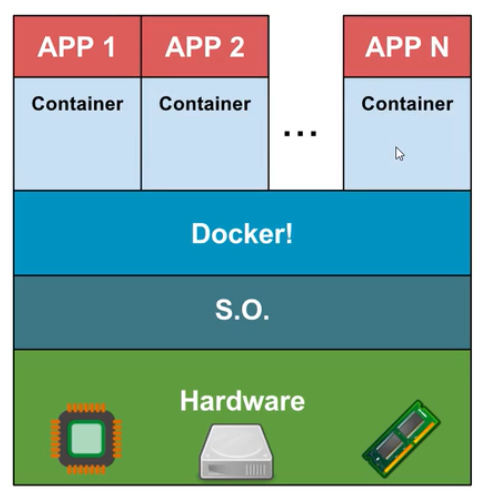
Com os containers temos:

* Melhor **controle** sobre o uso de cada recurso (CPU, Memória, Rede, etc)
* Agilidade para **criar/remover/reiniciar** containers (não prejudicando as outras Apps de outros containers)
* Maios facilidade para trabalhar com **diferentes versões de linguagens/frameworks**
* **Mais leves** que as VM’s

1. **Docker**

Nasceu como uma *Paas – Platform as a Service* , assim como o Azure, Heroku.

A Docker Inc. utilizava a **AWS**, introduzindo o Docker nas máquinas virtuais da AWS para administrar os containers na máquina virtual. Sendo um intermediário entre os **Containers** e o **SO**.



**Docker Engine**

Tecnologias de containers para prover ferramentas modernas para **publicar** e **rodas** as Apps

**Docker Compose**

Ferramenta para orquestrar múltiplos containers

**Docker Swarm**

Ferramenta para colocar múltiplos Docker’s Host’s para trabalharem em Cluster

**Docker Hub/Docker Storm**

Repositório com diferentes imagens para seus containers

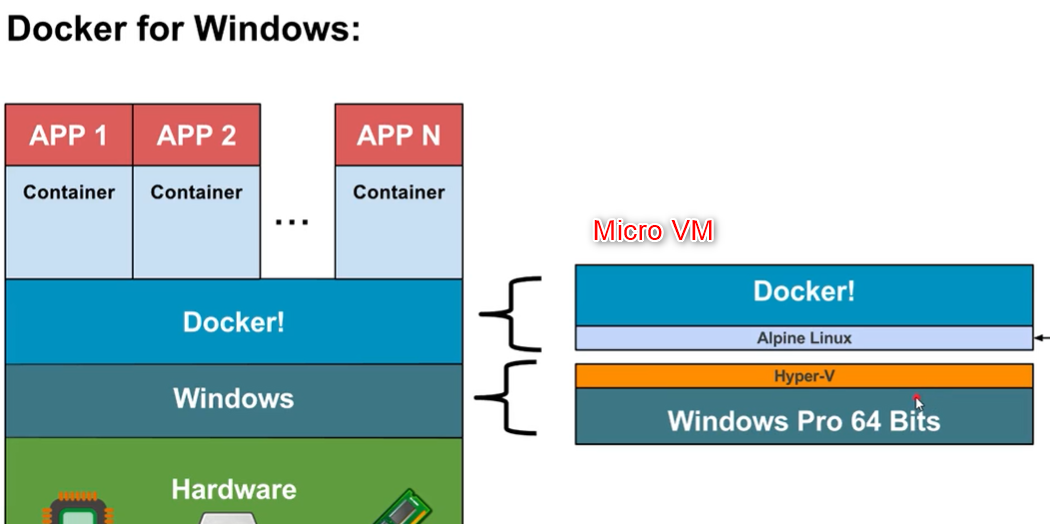
**Docker Machine**

Ferramenta que nos permite instalar e configurar em Hosts Virtuais

**Instalando no Windows**

Requisitos:

* Arquitetura 64 bits
* Versão **Pro**, **Enterprise** ou **Education**.
* Virtualização habilitada



**Hyper-V:** Permite executar vários sistemas operacionais como máquinas virtuais no Windows

* Permitindo assim que o Alpine Linux (Micro VM) seja executado

**Instalação alternativa no Windows**

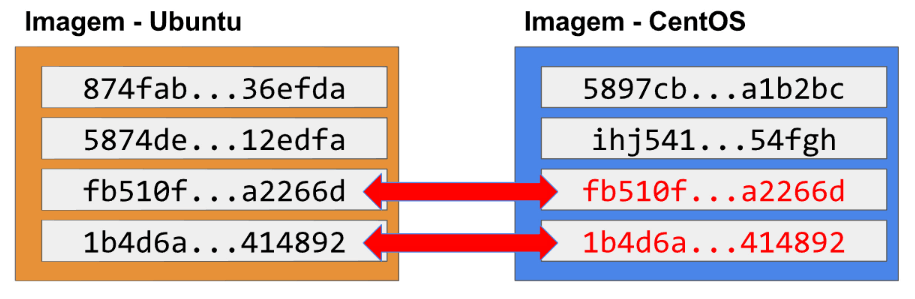
Precisamos garantir que o nosso Windows seja **64bits** e que ele tenha a virtualização habilitada.

* O **Docker Toolbox** vai instalar tudo que é necessário para que trabalhemos com o Docker em nosso computador, pois ele irá instalar também a **Oracle VirtualBox**, a máquina virtual da Oracle que vai permitir executarmos o Docker sem maiores problemas.
* A diferença é que, quando trabalhamos com o **Docker for Windows**, podemos utilizar o terminal nativo do Windows, já no **Docker Toolbox**, ele instalará o **Docker Machine**, que deverá ser utilizado no lugar do terminal nativo do Windows.

1. **Layered Filesystem**

Toda imagem que baixamos é composta por **uma ou mais camadas**

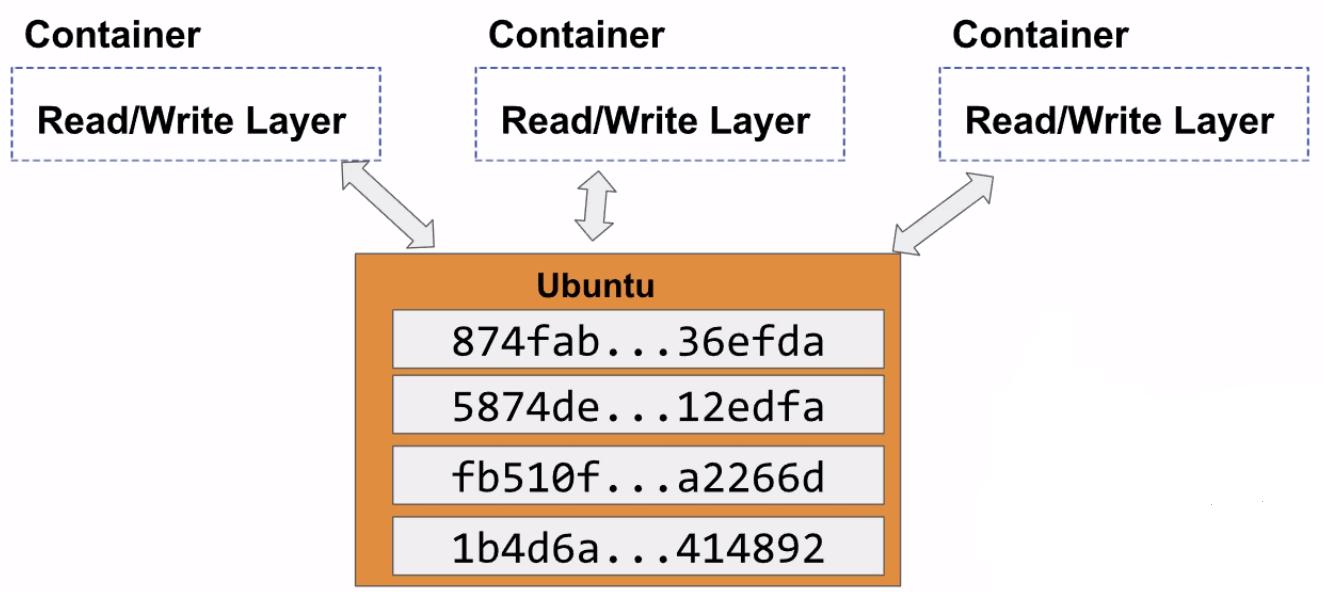
Essas camadas podem ser **reaproveitadas** em outras imagens. Se alguma camada é compartilhada entre si, o Docker baixará somente as camadas diferentes



No exemplo seriam baixadas apenas as 2 primeiras camadas do CentOS

Uma outra vantagem é que as camadas de uma imagem são **somente leitura**. Acima das camadas da imagem é criada uma nova camada que é possível **Ler e Escrever** nela

* Containers representam uma **instância** de uma imagem



Isso nos traz **economia de espaço**, já que não precisamos ter uma imagem por container

1. **Usando Volumes**

Quando escrevemos em um container, assim que ele for removido, os dados também serão. Mas podemos criar um local especial dentro dele, e especificamos que esse local será o nosso **volume de dados**.

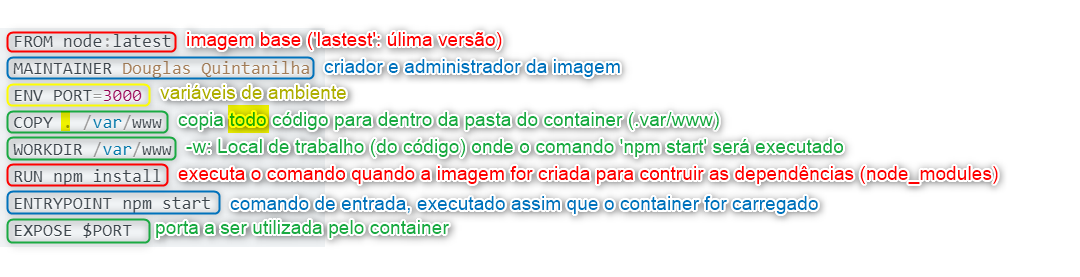
Quando criamos um volume de dados, o que estamos fazendo é apontá-lo para uma pequena pasta no **Docker Host**. Os containers até podem ser excluídos, mas a **Docker Engine** continua intacta.

Também é possível configurar o container para apontar o volume para uma pasta específica do computador físico.

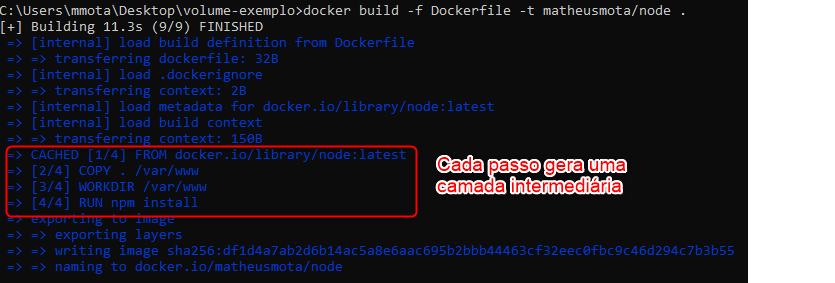
* Com isso, é possível utilizar o container como um **ambiente de desenvolvimento**
  + Podemos mapear um container **node**, por exemplo, apontando para o código na máquina local. Assim, ao executar o container apontando para o código, ele executará todo ambiente no container.
    - Isso minimiza o **conflito entre os ambientes**

1. **Dockerfile – Gerar a própria imagem**

Construindo uma imagem para outros desenvolvedores poderem ter o ambiente de desenvolvimento em sua própria máquina.



O Docker aproveita sua estrutura em camadas para criar **camadas intermediárias** de acordo com os comandos no Dockerfile:



Reaproveitando camadas iguais, conforme visto em **4. Layered Filesystem**

Para subir a imagem no **Docker Hub**:

* Crie um Login no Docker Hub
* Executar: docker login
* Executar: docker push matheusmota/node (estando na raiz do projeto)

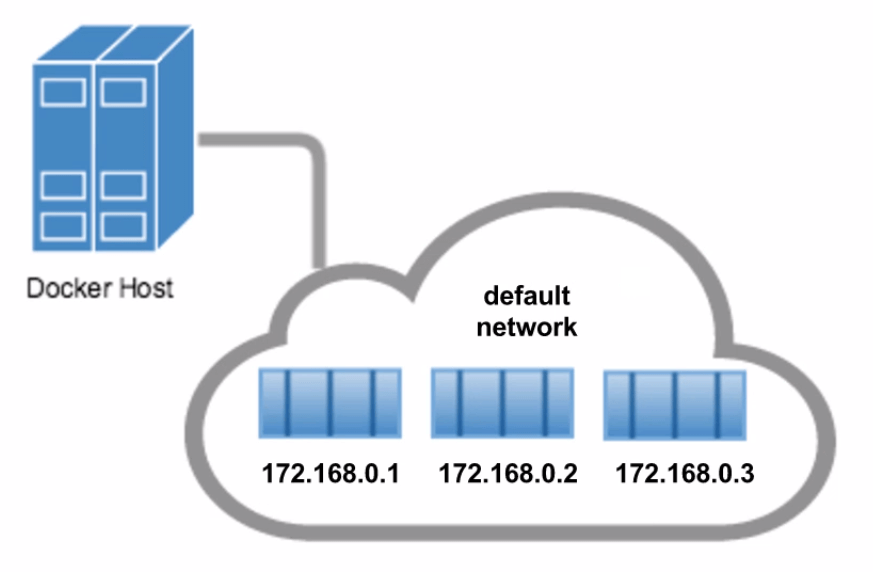
1. **Comunicação entre containers**

Uma aplicação é composta por diversas partes: Load Balancer/Proxy, a aplicação, um banco de dados, etc.

Ao trabalhar com containers é comum separar estas partes em containers específicos, deixando cada container com sua **responsabilidade única**.

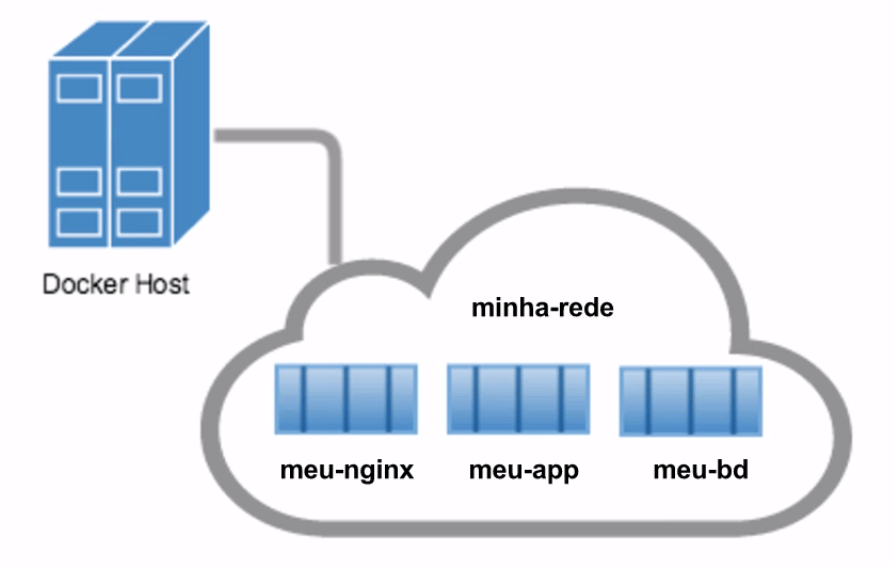
Para que estas partes se comuniquem, é necessário que estejam em uma rede se comunicando entre si.

No Docker já existe uma **“default network”**:



Todo container que é criado, ele recebe um IP e é criado dentro dessa rede padrão que o Docker cria automaticamente.

O problema é que esse IP é atribuído automaticamente pelo Docker e nesta rede o Docker não permite que seja atribuído um **hostname** ao container, apenas atribuir um nome a ele. Para resolver é necessário criarmos **nossa própria rede**, e fazer com que a comunicação entre eles seja feita pelo nome:



1. **Docker Compose**

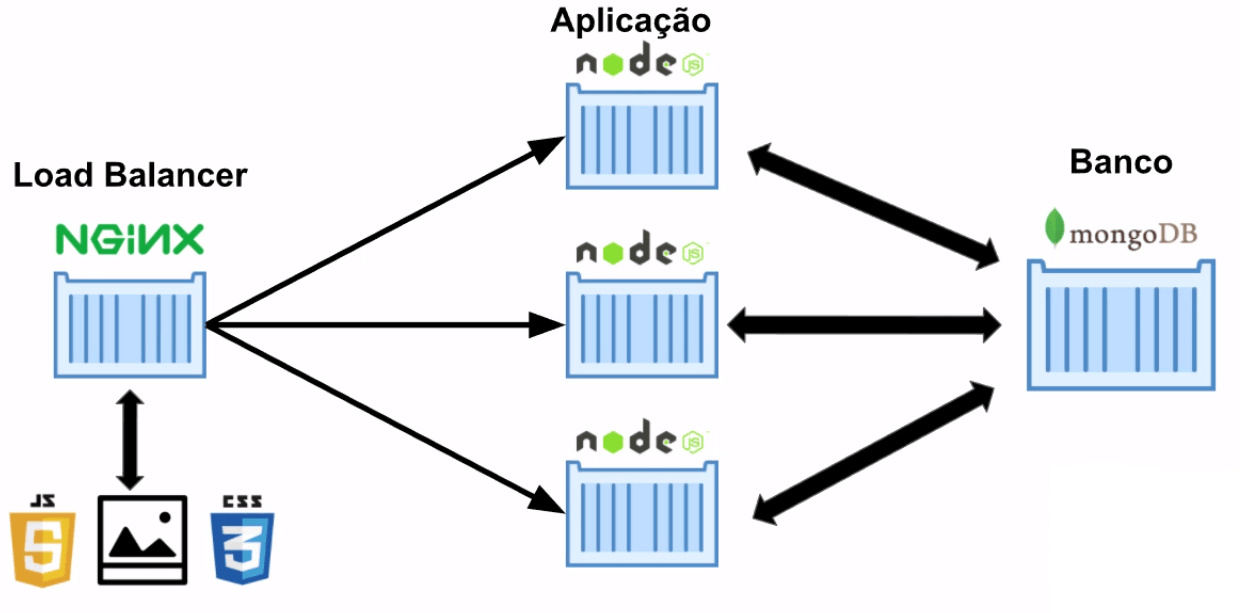
Tecnologia que ajuda a orquestrar a **execução de múltiplos containers**.

* docker-compose**.yml**
  + Todos os passos que vão ser seguidos ao subir uma aplicação
  + Todo o processo de **build** fica descrito nesse arquivo

**Porque utilizá-lo?**

* Para subir uma aplicação é necessário digitar vários comandos, com várias flags e na ordem certa (subir primeiro o banco de dados, depois a aplicação, ...)
* Ou seja, é um trabalho **manual** e suscetível a erros
* Se precisarmos subir os containers para preparar um ambiente **rapidamente**, seria difícil com todo esse trabalho manual

**Exemplo:**



Neste exemplo temos uma aplicação distribuída em **3 containers** para segurar o tráfego da aplicação e **distribuir a carga**.

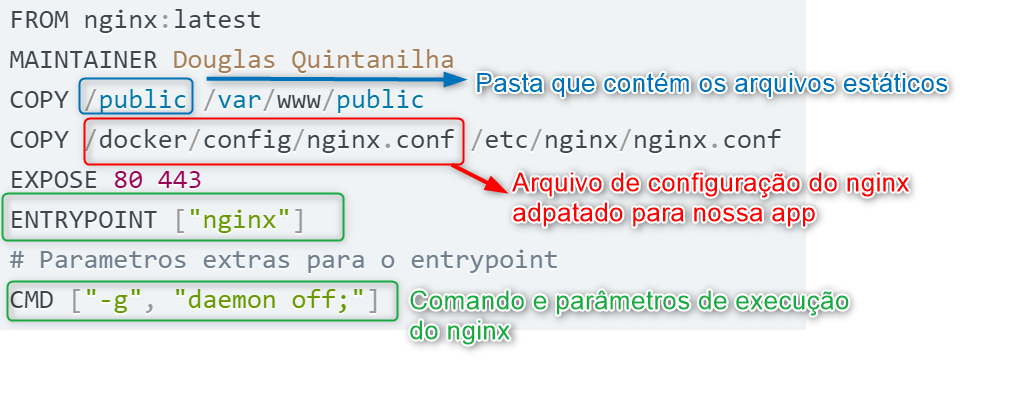
Se temos 3 aplicações rodando, não podemos ter endereços diferentes, para isso é utilizado um ***Load Balancer*** em um outro container para fazer a distribuição das cargas quando tiver muito acesso.

* O ***Load Balancer*** recebe as requisições e distribui para uma das aplicações. É também utilizado para servir os arquivos estáticos (como imagens, .css e .js), de modo que a aplicação fique responsável somente pela lógica de negócio

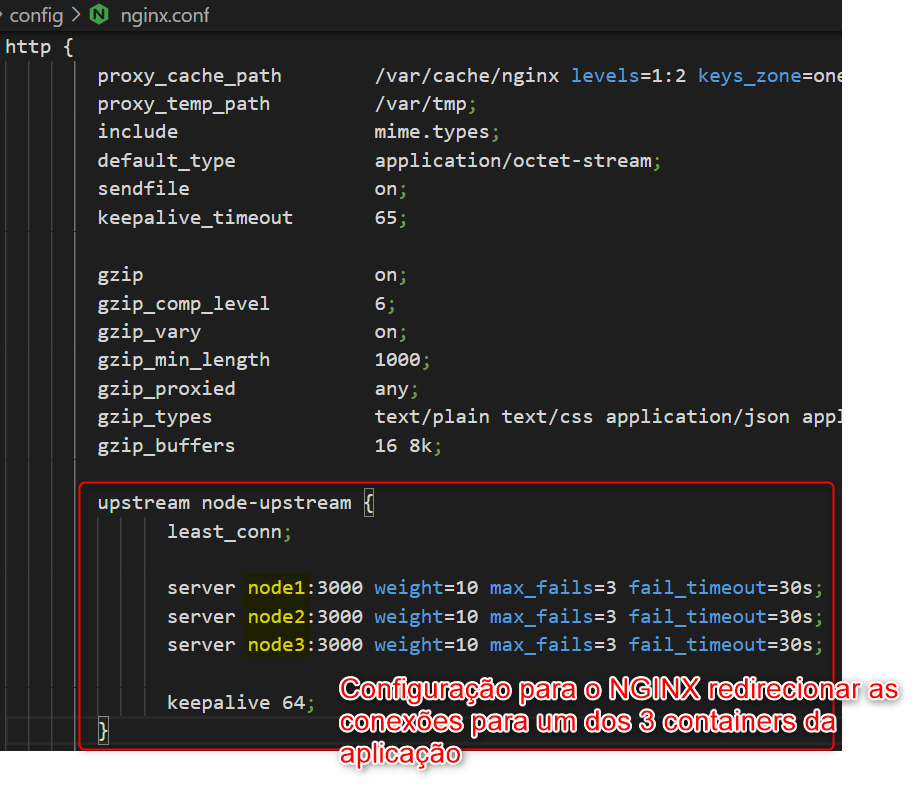
Se seguirmos o exemplo acima teremos 5 containers para serem executados, e numa ordem determinada.

**Configurando o NGINX**

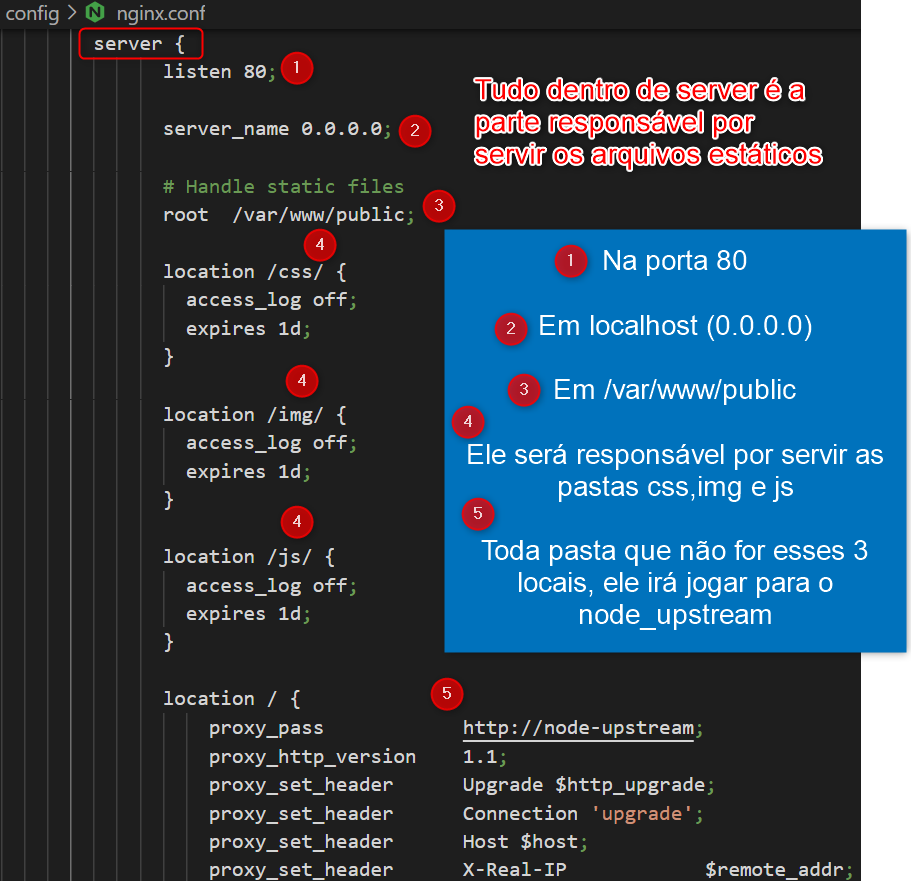
Existe a imagem padrão do NGINX, porém como vamos lidar com algo mais específico, como lidar com os arquivos estáticos e balancear a aplicação em 3 containers, vamos criar a própria imagem:



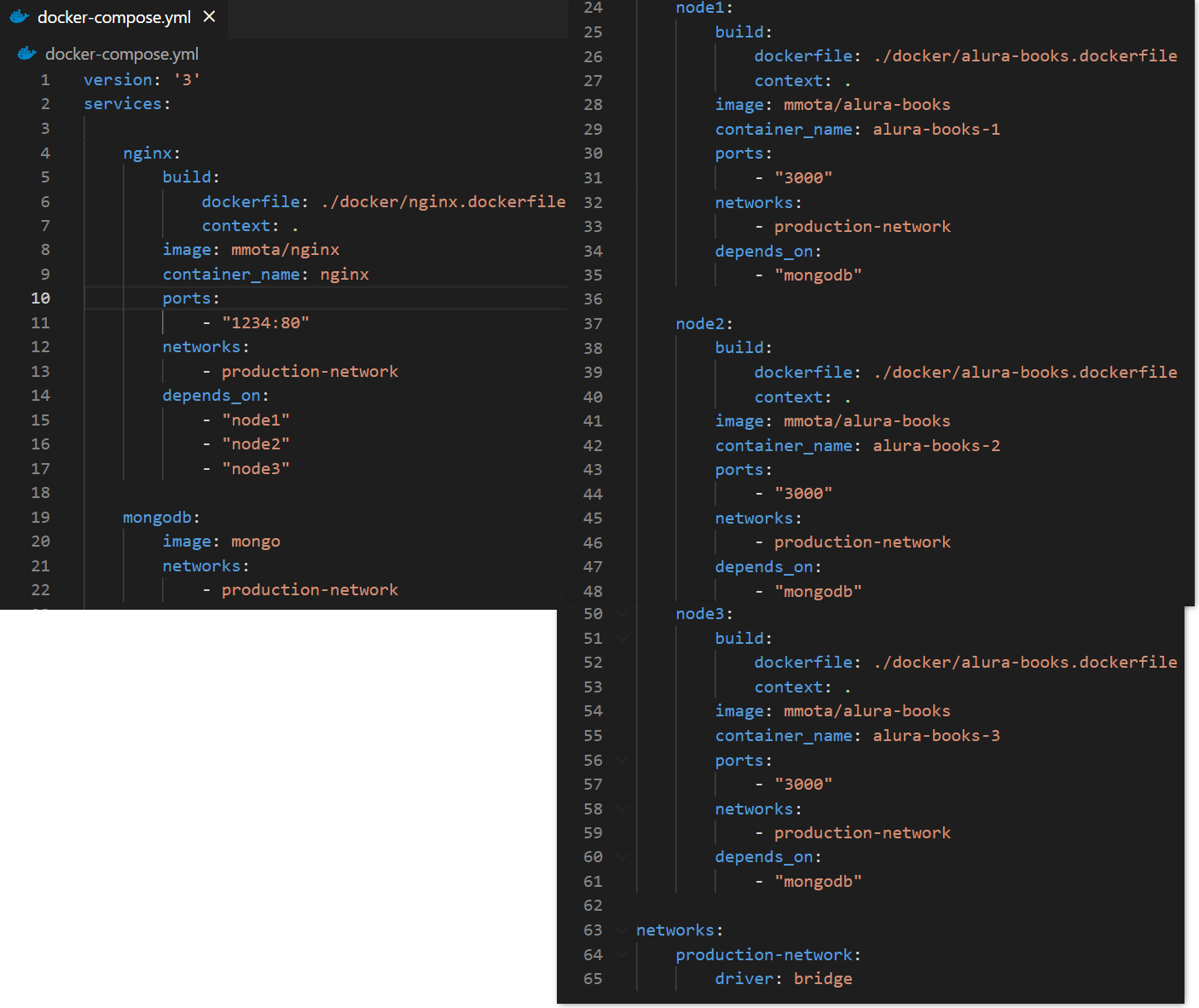
Entendo o arquivo **nginx.config**:



O redirecionamento acontece de forma circular, primeira conexão para o primeiro container, segunda conexão para o segundo container, terceira conexão para o terceiro container, na quarta, inicia o ciclo novamente.



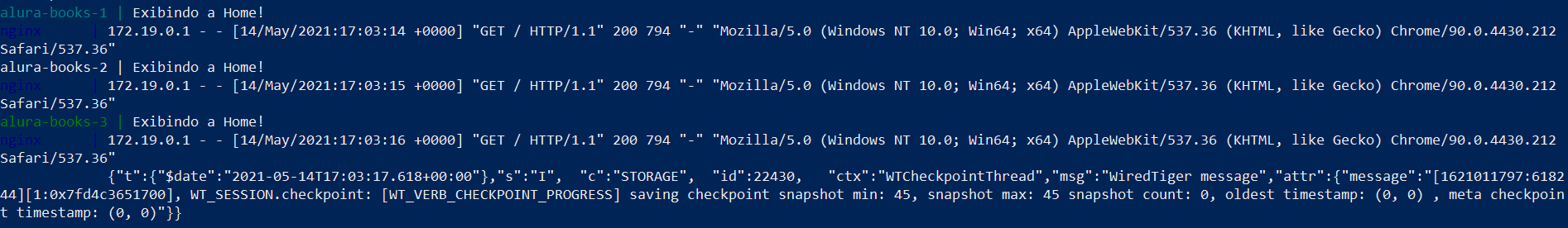
**Criando o arquivo .yml**

****

* **Version**: versão do arquivo **.yml**
* **Services**: todos os containers que serão executados
* **Nginx, mongoDb, node1, node2 e node3**: nome do serviço
* **Build**: utilizado quando é necessário buildar um dockerfile existente
  + **Dockerfile**: Local do arquivo
  + **Contexto**: No exemplo o contexto é a pasta local
* **Image**: No caso do mongo, como é uma imagem existente no Docker Hub, é passado o nome da imagem.
  + Já no caso das imagens construídas por um dockerfile, é passado o nome da imagem
* **Container\_name:** nome do container
* **Ports:**
  + “1234:80” = Porta Local: Porta exposta no Container (de acordo com a app)
  + “3000” = Porta a ser exposta no container (de acordo com a app)
* **Networks:** determina a rede que o container deve pertencer
* **Depends\_on:** determina a ordem de criação/exlcusão dos containers
  + **NGINX >> Apps >> Mongo**
* **Networks:** criar a própria rede
  + **Production-network**: nome que demos à rede
  + **Driver**: especifica o driver a ser utilizada na rede a ser criada

**Funcionamento do Load Balance**

Cada requisição vai para um conatiner

****

**Docker e Microsserviços**

Trabalhar com uma arquitetura de microsserviços gera a necessidade de **publicar o serviço de maneira rápida, leve, isolada** e vimos que o Docker possui exatamente essas características! Com Docker e Docker Compose podemos criar um ambiente ideal para a publicação destes serviços.

O Docker é uma ótima opção para rodar os microsserviços pelo fato de **isolar os containers**. Essa utilização de containers para serviços individuais faz com que seja muito simples gerenciar e atualizar esses serviços, de maneira automatizada e rápida.

**Docker Swarm**

Agora vou ter vários serviços rodando usando o Docker. E para facilitar a criação desses containers já aprendemos usar o Docker Compose que sabe subir vários containers. O Docker Compose é a ferramenta ideal para **coordenar a criação dos containers**, no entanto para melhorar a **escalabilidade e desempenho** pode ser necessário criar muito mais containers para um serviço específico. Em outras palavras, agora gostaríamos de criar **muitos containers aproveitando várias máquinas** (virtuais ou físicas)! Ou seja, pode ser que um microsserviço fique rodando em 20 containers usando três máquinas físicas diferentes. Como podemos facilmente subir e parar esses containers? Repare que o Docker Compose não é para isso e por isso existe uma outra ferramenta que se chama **Docker Swarm**.

Docker Swarm facilita a criação e administração de um cluster de containers**.**