## **Revisando tudo o que foi visto**

Chegamos ao final do nosso curso de **Docker** e agora faremos uma revisão de tudo que vimos até aqui. Porém, antes de prosseguirmos, vejamos alguns dados e fatos importantes acerca do Docker.

## **O que é o Docker?**

No início, o Docker foi um projeto particular de uma empresa chamada dotCloud. Ele resumia-se em ser um serviço de PaaS, uma ideia de plataforma como serviço. Além disso o Docker é Open Source e escrito na linguagem GO. Tal projeto teve início em 18/01/2013 e sua primeira versão saiu em 25/03/2013.

## **Voltando no tempo... Virtualização de hardware e de processos**

Na lista abaixo é possível observar diversos acontecimentos em relação à virtualização de hardware, estando os mais importantes em itálico:

* 1966-1972: IBM CP/CMS
* 1989: Insignia SoftPC
* 1997: Connectix VirtualPC
* *1999: VMWare Workstation*
* *2002: Xen*
* 2001: IBM AIX LPAR
* *2006: Amazon EC2*
* 2007: Sun Logical Domains
* 2007: Linux KVM
* 2007: InnoTek VirtualBox
* 2008: MS Hyper-V

Paralelo a esse desenvolvimento temos a virtualização de processos:

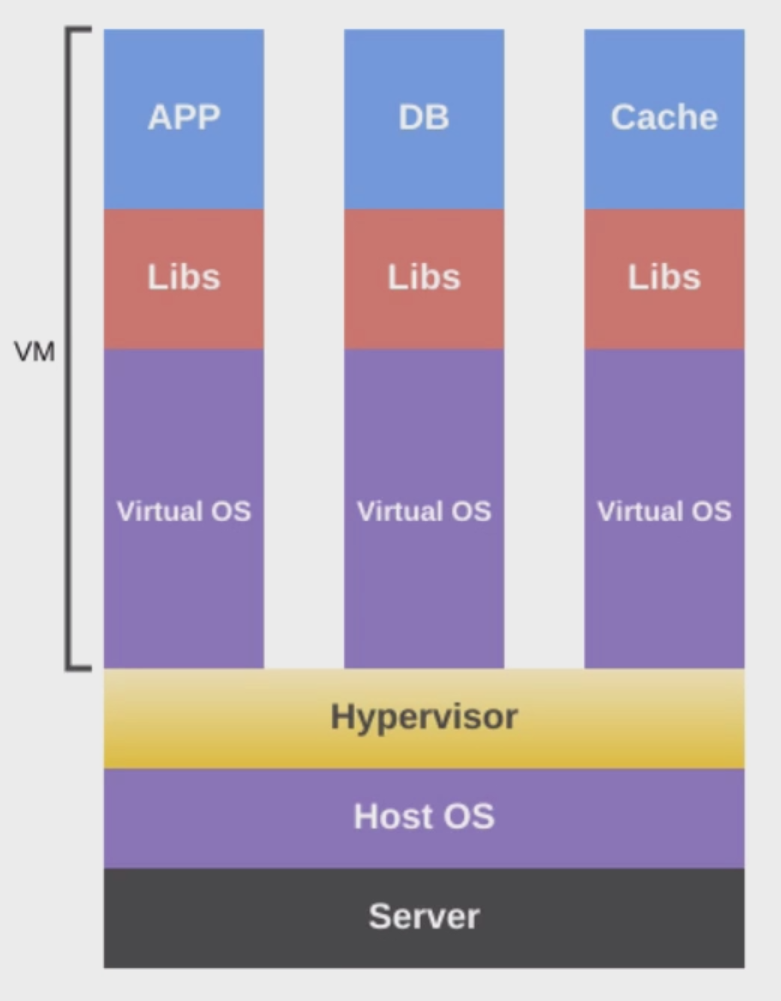
* *1979-1982: UNIX chroot*
* 1998: FreeBSD jail
* 2001: Parallels Virtuozzo
* 2001: Linux-VServer
* 2005: Solaris Containers
* 2005: OpenVZ
* *2008: Linux LXC*
* *2007-2008: (Heroku)*
* *2013:* ***Docker***

## **Diferenças entre uma máquina virtual (VM) e um container**

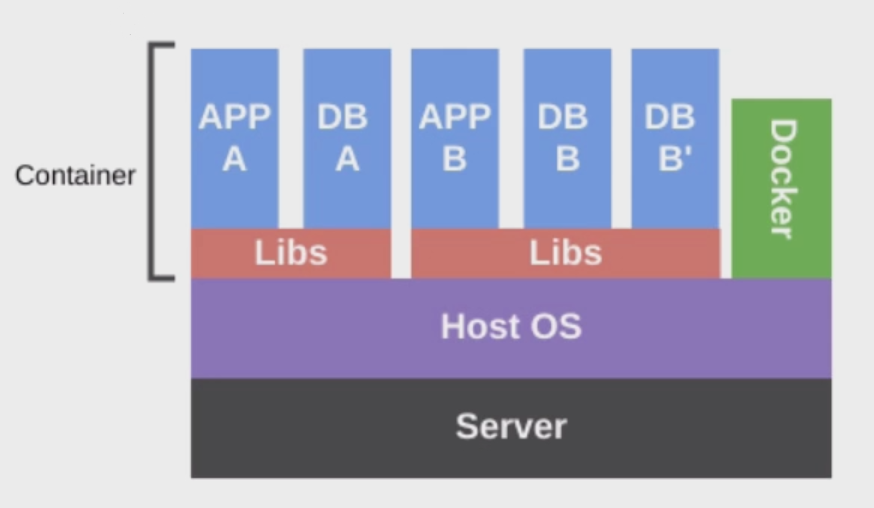
Em uma máquina virtual temos

* **Server**: é o hardware de nosso servidor;
* **Host OS**: sistema operacional hospedeiro, responsável por controlar as máquinas virtuais
* **Hypervisor**: pode ser descrito como a nossa máquina virtual propriamente dita, é quem gerencia as máquinas virtuais

Sempre que quisermos criar uma máquina virtual (VM), precisamos carregar todo o Sistema Operacional (OS) e suas bibliotecas para, por último, rodar nossa aplicação. O mesmo vale para um sistema de Banco de dados (DB) ou de Cache. No exemplo abaixo levantamos três máquinas virtuais em que para cada uma levantamos três OS. Isso gera sobrecarga para o servidor.



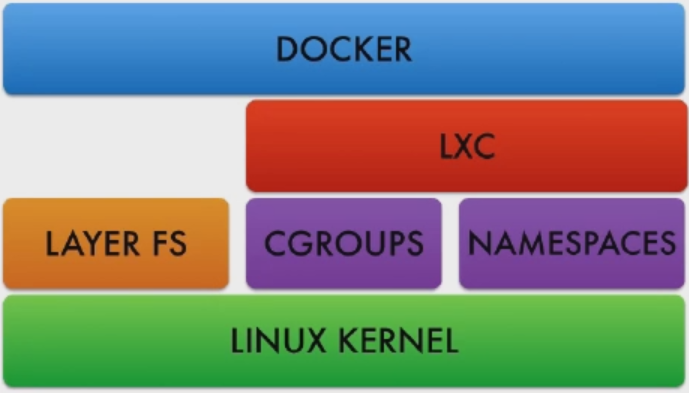
Em contrapartida, existe a ideia de containers. Da mesma forma que a VM, também existem um Servidor e um Host OS. Os containers andam em paralelo, podendo ser Linus Container, Docker, etc. Para cada container criado, o Docker irá gerenciar o Kernel do Host OS e suas bibliotecas e os irá compartilhar com os processos isolados, que são os containers propriamente ditos. No cenário abaixo, verificamos que há uma baixa sobrecarga, porque existe o reaproveitamento de todos os recursos (bibliotecas, etc) do Host OS.



Perceba que nas VMs é necessário levantar um OS inteiro, enquanto que usando os containers esse processo funciona de forma compartilhada, o que os fazem mais rápidos. Mas como funciona essa otimização? Daremos uma olhada por dentro.

## **Arquitetura dos containers**

Na arquitetura dos containers - do Docker, em especial - temos na camada mais baixa o Linux Kernel do OS. O Cgroups e o Namespaces são responsáveis por isolar e controlar os recursos que cada container vai utilizar durante sua existência. Temos também o sistema de gerenciamento de camadas, o Layer FS, para a construção das nossas imagens e uma interface controladora de container - podendo ser LXC ou Lib Container do próprio Docker. E na camada superior temos o Docker.

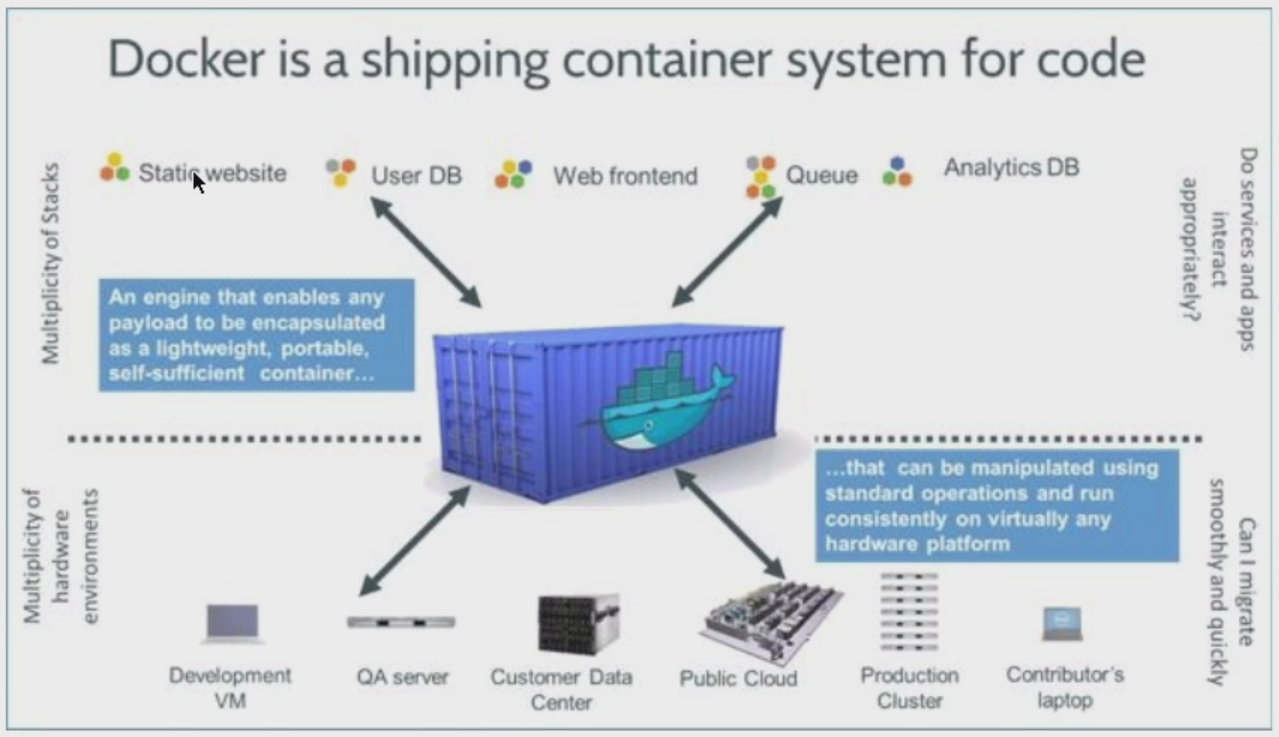


## **Quais problemas o Docker resolve?**

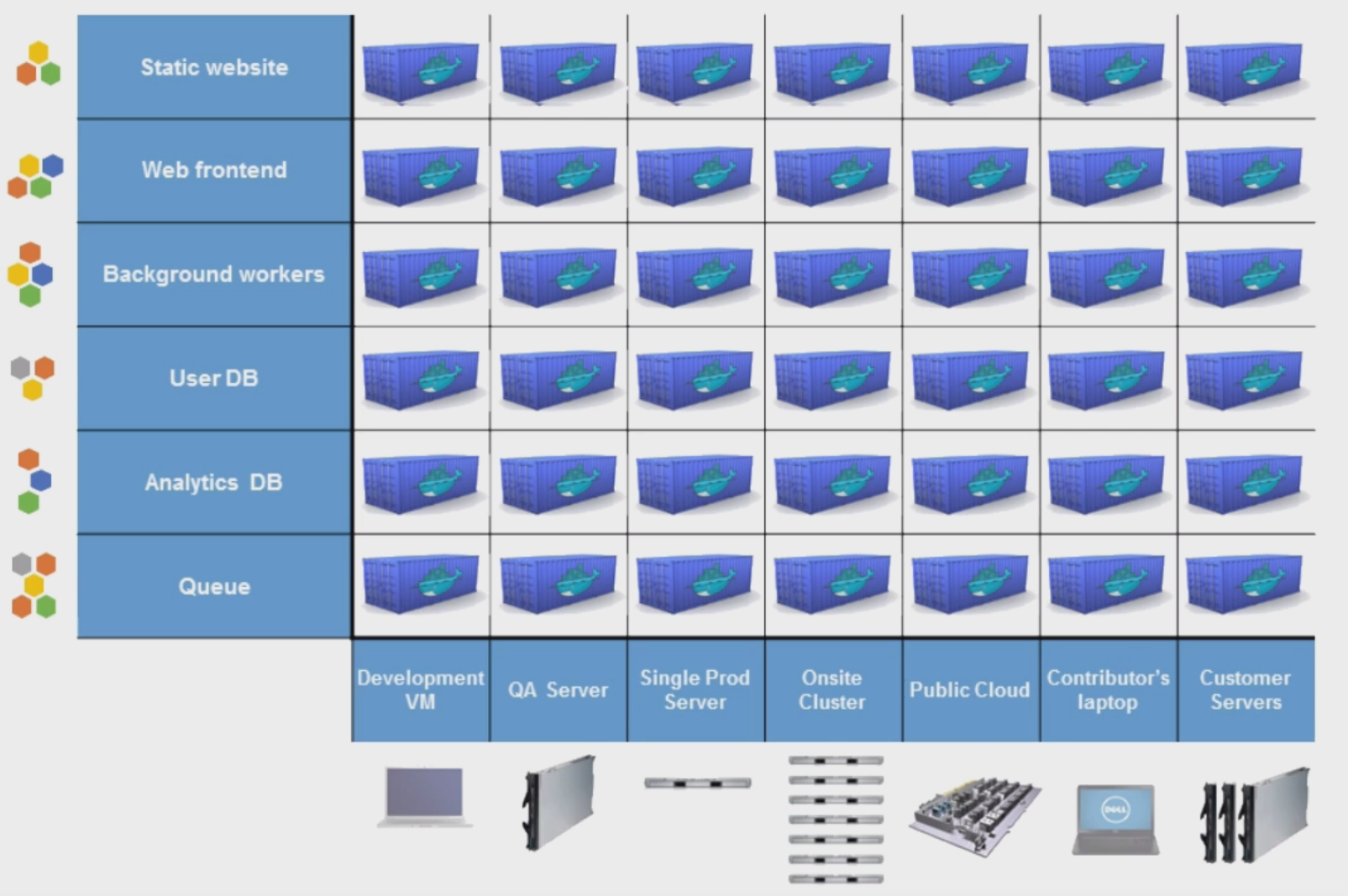
Temos que nos preocupar em como configurar cada parte da nossa aplicação dependendo do ambiente:



Pensando nisso, se a ideia de containers for aplicada, teremos este cenário:



O elo comum entre qualquer ambiente - seja ele de desenvolvimento, de teste, de produção - e todas as partes que compõe nossa aplicação é o container. Então, preenchendo as interrogações da tabela acima, teremos a seguinte solução:



O container utilizado em um ambiente de desenvolvimento será idêntico ao utilizado no servidor customizado, ou no notebook do cliente! E o Docker funciona no Linux (LXC e libcontainer), no OS X (boot2docker e kitematic) e no Windows (boot2docker)!

## **Imagens**

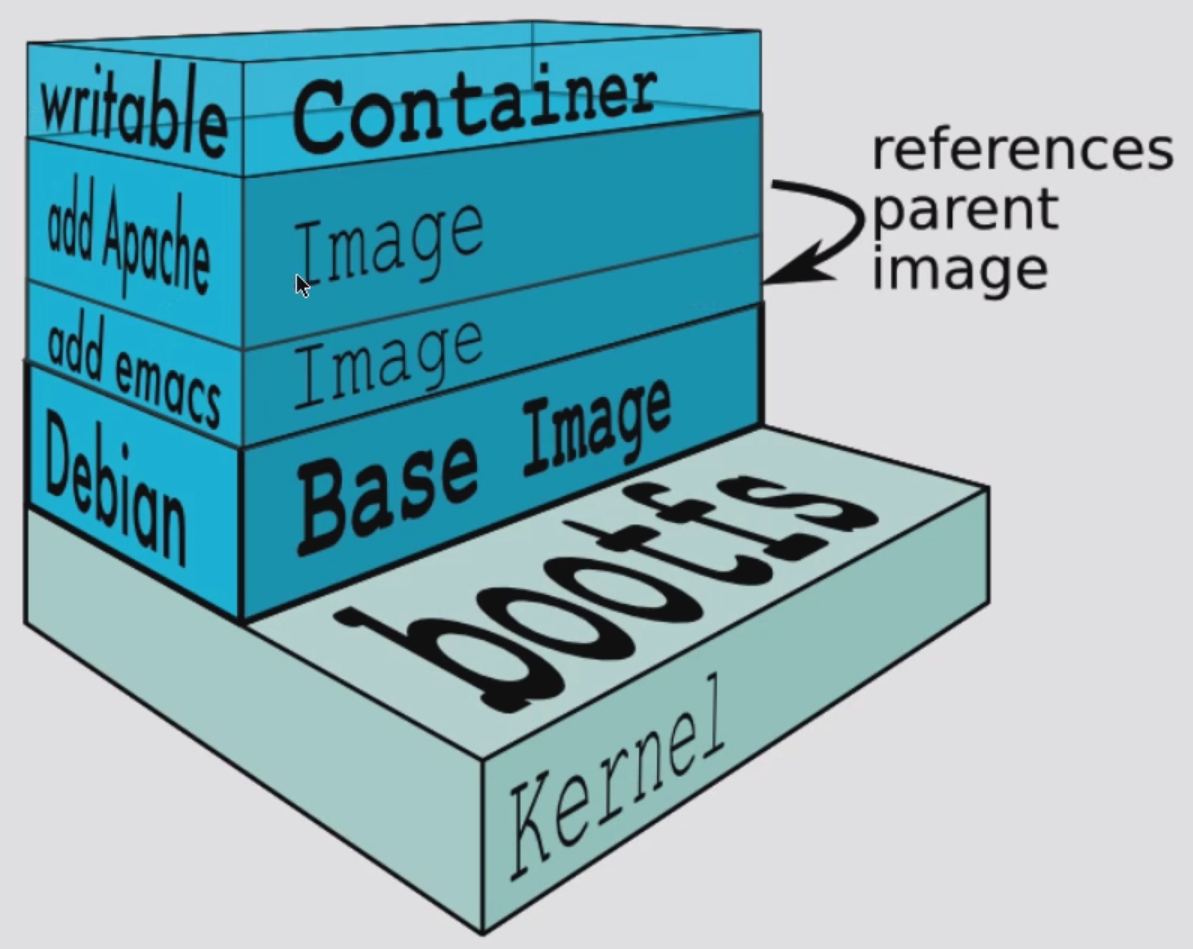
* São templates read only
* Fazem uso do sistema UFS
* Contém os arquivos da sua app
* Criadas por Commit ou Dockerfile

## **Containers**

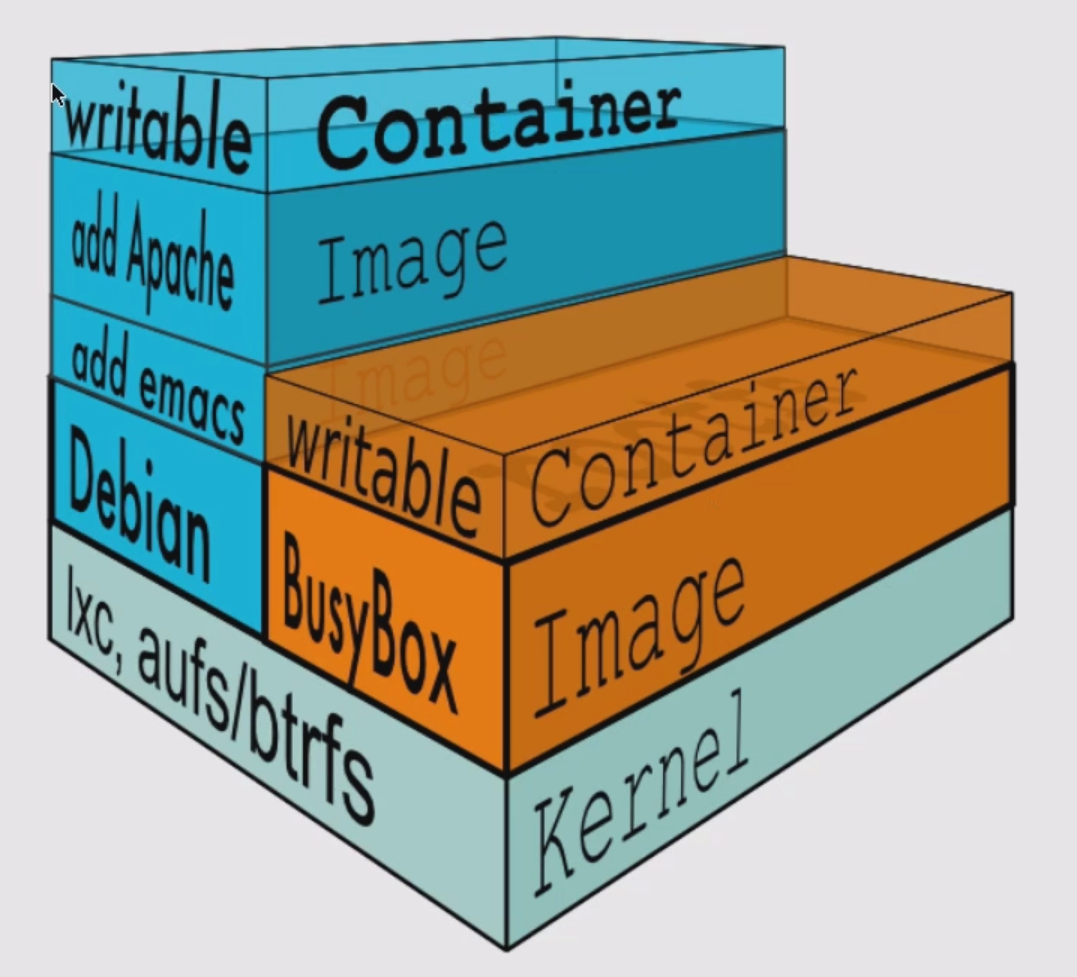
* Dependem de uma imagem
* Geram novas imagens
* Estabelecem conectividade com o host e outros containers
* Possuem uma execução controlada - CPU, RAM, I/O, etc
* São descartáveis ou persistentes

## **Camadas de imagens**

Digamos que criamos um container com permissão de escrita. Quando isso acontece, podemos instalar por exemplo o *emacs* e em seguida o Apache. Cada instalação dessa representa uma camada em cima da imagem base, a qual é read only e não sofre alterações, apenas acréscimos de camadas. Esse processo nos permite fazer alterações removendo ou adicionando camadas.



O uso dos Cgroups e Namespaces permite que façamos isso com vários containers simultaneamente e isolados. Poderíamos ter dois ou mais containers trabalhando ao mesmo tempo criando camadas em cima de suas imagens base sem um interferir o outro.



## **Inicializando containers**

* Opção **run** com interatividade:
* **$ docker run -it nome\_da\_imagem [comando]**
* Executando em background:
* **$ docker run -d nome\_da\_imagem**
* Eliminando após o uso:
* **$ docker run --rm -it nome\_da\_imagem [comando]**

## **Criando imagens**

* Iniciar container interativo
* **$ docker run -it ubuntu**
* Realizar alterações:
* **$ sudo apt-get update && apt-get install -y nome\_do\_pacote**
* Encerrar e commitar
* **$ docker commit [id] nome\_da\_imagem**
* Dockerfile
* **$ docker build -t nome\_da\_imagem**
  + Dentro do Dockerfile:
  + FROM [imagem da qual será herdado]  
      
     RUN [instalação de pacotes]  
      
     EXPOSE [abertura de portas]  
      
     CMD [execução de comandos]
* Gerando nossas próprias imagens com a opção build seguida de user/serviço e dockerfile
* **$ docker build -t user/serviço [caminho]**

## **Utilizando imagens prontas**

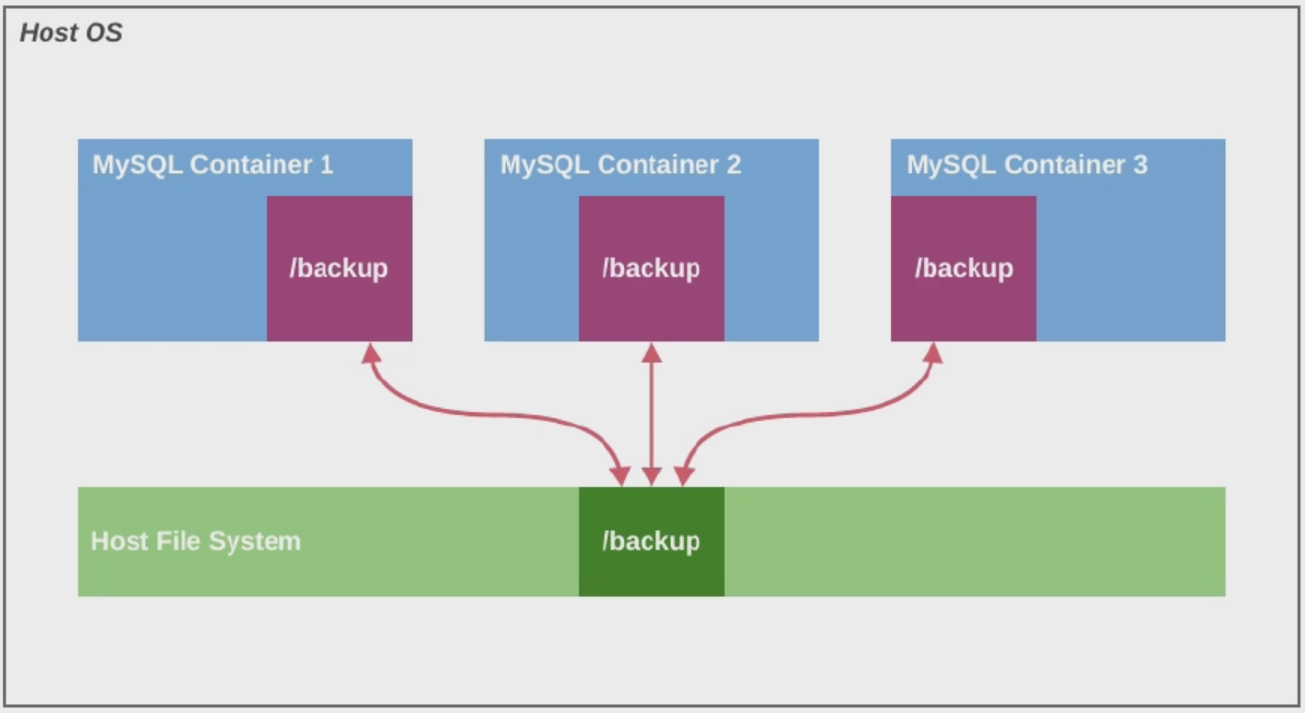
Podemos buscar em repositórios oficiais e não oficiais em [Login](https://hub.docker.com/login/)

**$ docker login**

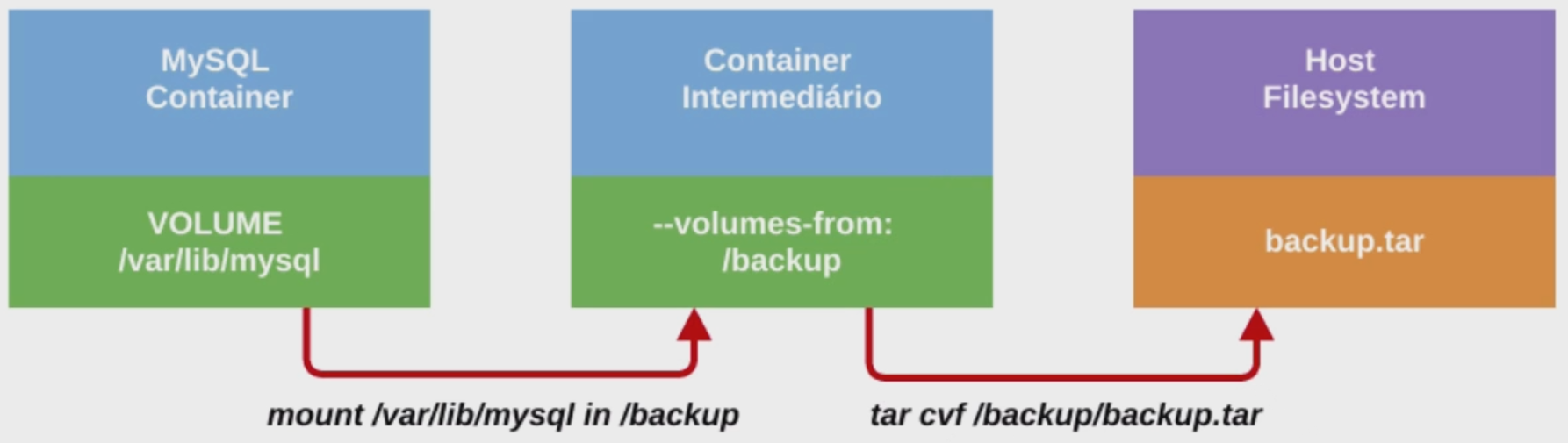
* Baixando as imagens
* **$ docker pull nome\_da\_imagem**
* Buscando imagens
* **$ docker searche nome\_da\_imagem**
* Enciando imagens
* **$ docker push user/serviço**

## **Volumes**

Volumes são áreas reservadas dentro do container para estabelecer comunicação o Host File System do nosso Host Local. O mapeamento dessas áreas reservadas nos permite ter uma persistência de dados de dentro dos containers para o Host Local e vice versa.



Aqui podemos observar um exemplo do uso de volumes:



Nosso curso de Docker terminou. Esperamos que vocês tenham gostado! Nos vemos no próximo!