Containers

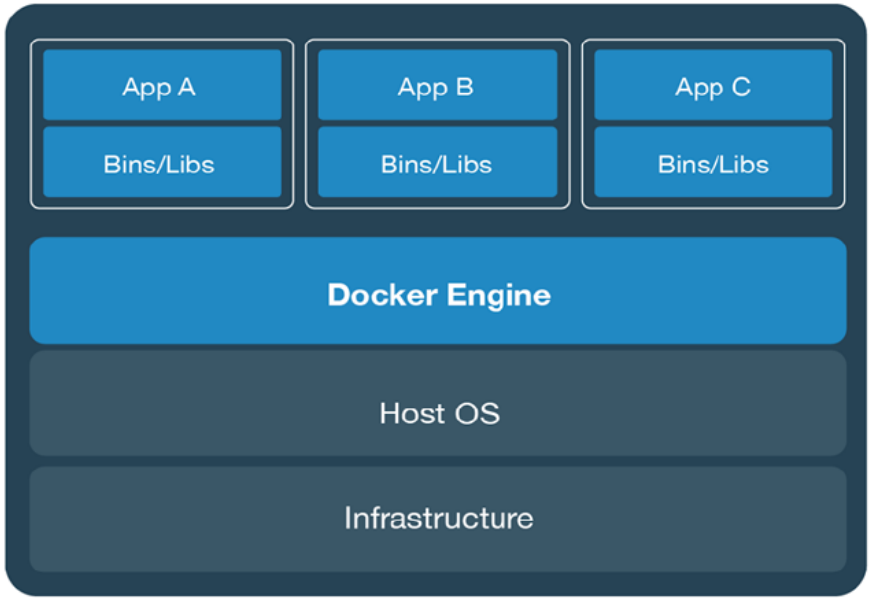
"Never, ever, think outside the box."

VM's vs Containers

- Máquinas Virtuais ao nível do SO (e.g. Xen)
  - Não disponibilizam virtualização completa do hardware
    - Apenas os serviços do SO são virtuais
    - *Host Kernel*: virtualiza os seus serviços de forma isolada ao diferentes inquilinos
- **Container**: isola o ambiente de execução de um ou mais processos

- A considerar:
  - Isolamento/Visibilidade: limitar o que pode ser visto pelos inquilinos (*tenants*)
  - Controlo de recursos: limitar o consumo de recursos
  - Portabilidade: poder reconstruir o mesmo ambiente em múltiplos *hosts*

Virtualização baseada em Containers



- Virtualização com base em containers depende dos mecanismos do SO para atingir o isolamento de aplicações.

Suporte do SO

- Linux não tem qualquer suporte para containers
  - A *kernel* Linux não conhece o conceito de container
- Containers são um conjunto de configurações que definem um ambiente que é aplicado a um processo (e seus filhos)
  - Namespaces: isolamento e virtualização de cada inquilino
  - Cgroups: controlo de recursos de cada inquilino (CPU, RAM, IO)

Tecnologias subjacentes

- Namespaces
  - (mnt, pid, net, ipc, uts/hostname, user ids)
- Cgroups
  - (cpu, memória, disco, I/O – gestão de recursos)
- AppArmor, SELinux
  - (segurança/controlo de acesso)
- Seccomp
  - Isolamento de computação
- CHROOT
  - Isolamento de sistema de ficheiros

Namespaces

- Mecanismo utilizado para isolar e virtualizar os recursos do sistema
  - Processos num *namespace* não conseguem ver os recursos remanescentes
    - Vêm o seu *namespace* como sendo todo o *host*
- Network *Namespace*
  - Mecanismo que permite criar recursos de rede quase independentes
    - Interfaces de rede, tabelas de *routing*
  - Uma interface de rede só pode pertence a um único *namespace*
- PID *namespace*
  - *Namespace* restrito com identificadores de processo privados
    - Processos do *host* são invisíveis

Cgroups

- Mecanismo usado para restringir (limitar, controlar) ou monitorizar a quantidade de recursos utilizada por “grupos de processos”
  - Os processos podem ser organizados em grupos, e desta forma controla-se o acesso aos recursos
  - Exemplo: Controlo de escalonamento de CPU
    - Limitar quantidade de tempo de CPU que um grupo de processos pode utilizar
  - Outros usos:
    - Memoria, I/O, pids, rede,….

Como construir um Container (1/2)

- Criar todos os *namespaces* e *cgroups*
  - Normalmente utilizando um *template*
- Criar um "disco virtual" para o container
  - Diretório que contém o sistema de ficheiros do container
  - Existe no *host* (é um diretório do mesmo)
- Fazer *chroot* para o sistema de ficheiros do container
  - O sistema de ficheiros precisa de ter todas bibliotecas/ficheiros necessários para executar o programa
- Iniciar o programa a *containerizar*
  - Este processo terá o PID 1 do container
  - Este processo pode montar procfs e outros pseudo sistemas de ficheiros
    - *Namespaces* podem ser usados para limitar a informação nestes pseudo sistemas de ficheiros

Como construir um Container (2/2)

- Devido ao *namespace* de rede, os processos *containarizados* não vêm as interfaces de rede do *host*
  - Mas também não é normal atribuir interfaces de rede reais a um namespace (teria impacto no *Host*)
- Rede em containers
  - Criar um par de interfaces ethernet virtuais: 2 interfaces, ligadas ponto-a-ponto, interligam *host* e container.

Ferramentas

- Criar containers à mão é difícil e trabalhoso
  - Número elevado de instruções para criar *namespace*, *cgroups*, *chroots*, interfaces, bridges
- Ferramentas em *User Space* facilitam a manipulação de containers:
  - LXC – Linux Containers
  - Docker
  - Singularity
  - Kubernetes

Portabilidade

- Objetivo primordial de um Container é replicar ambientes de execução
  - Mesmo ambiente (*namespace*, *chroot*, *cgroups*) pode ser usado em diferentes *hosts*
- É pois essencial a definição (comum) de um container
- Open Container Initiative (OCI) – https://www.opencontainers.org
  - Define standards para ferramentas de *userspace*
    - Especificações *runtime*: configuração, execução e ciclo de vida de um container
    - Especificação da imagem: como representar dados de um container

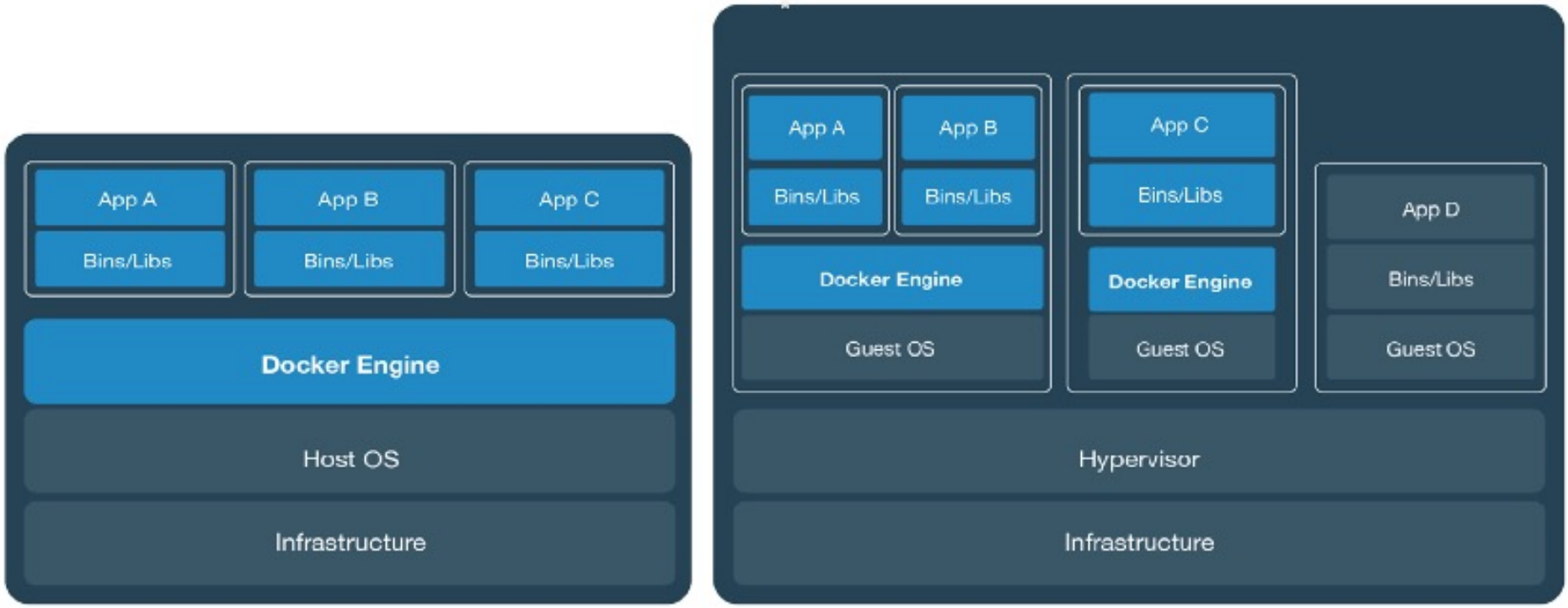
Docker

- Produto Comercial
  - Evolução do conceito do Solaris Containers e Linux Containers (LXC)
  - Libertado com licença OSS em 2013
- Conceito principal: Container
  - Construção com base na aplicação (desligado da infraestrutura)
  - Composto por múltiplas funcionalidade (*namespaces*) que se coordenam entre si
  - 1 Container – 1 Aplicação (que pode no entanto fazer *spawn*/*fork*)
- Containers embarcam tudo o necessário para correr uma aplicação em qualquer hardware
  - Configurações, dependências, dados auxiliares, etc.

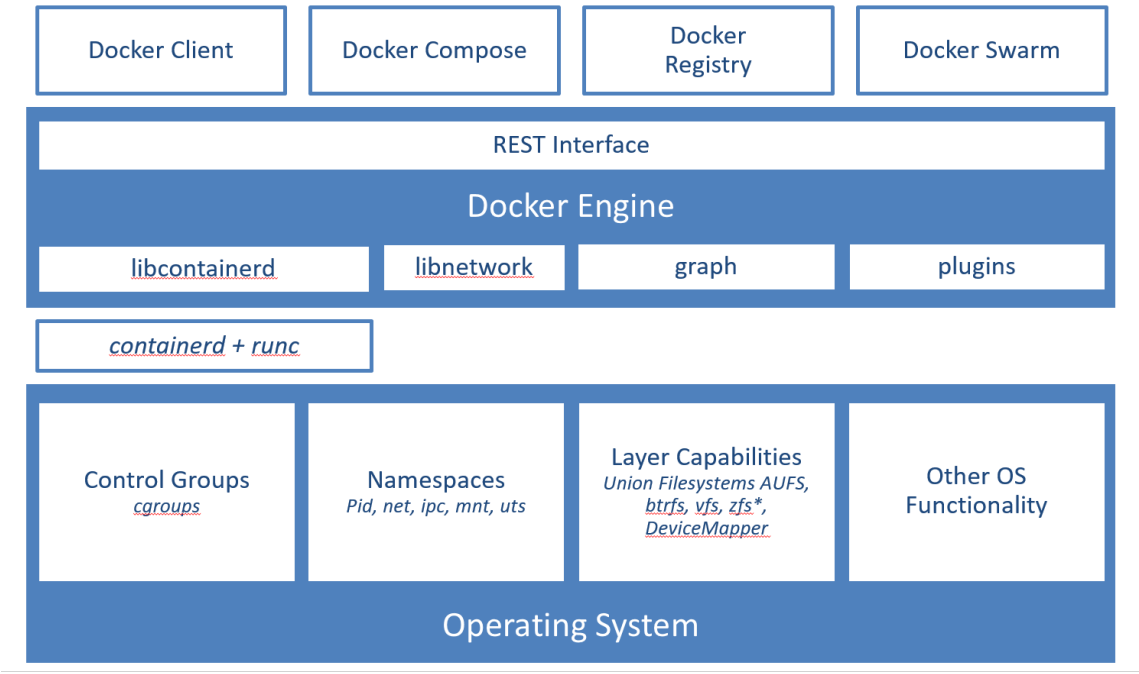
Conceitos Docker

- **Image**: os dados do container (aplicação, bibliotecas, imagens, etc)
- **Container**: A instância de uma aplicação em execução
  - Composto por uma ou mais imagens. Cada imagem acrescenta uma funcionalidade
- **Engine**: Software que executa os containers
- **Registry**: Repositório de imagens Docker
- **Control Plane**: Infraestrutura que gere os containers e as imagens

Containers podem correr em VM's



Arquitectura Docker



Docker Registry

- Repositório central que armazena e entrega imagens
  - Indexado por ***name*** e ***tag***
  - Pode ser privado ou publico (Docker Hub)
- Clientes fazem push de imagens para o *registry*
  - Cliente local
- Docker Engine faz pull das imagens e executa o container
  - Corre no servidor
- Camadas são *hashed* e indexadas permitindo reutilização
  - O pull de uma imagem apenas necessita de fazer download das camadas que não existem localmente

Workflow

Procurar uma imagem:

```
$ docker search nginx
```

Pull de uma imagem:

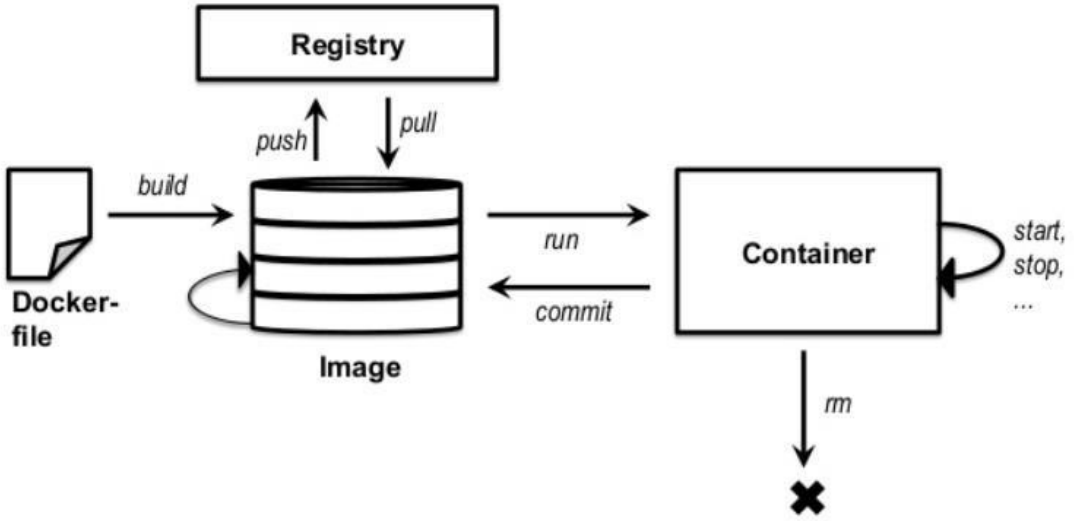
```
$ docker pull nginx
```

Listar imagens locais:

```
$ docker image ls
```

Correr um container:

```
$ docker run -d -name frontend-server nginx
```



Dockerfile

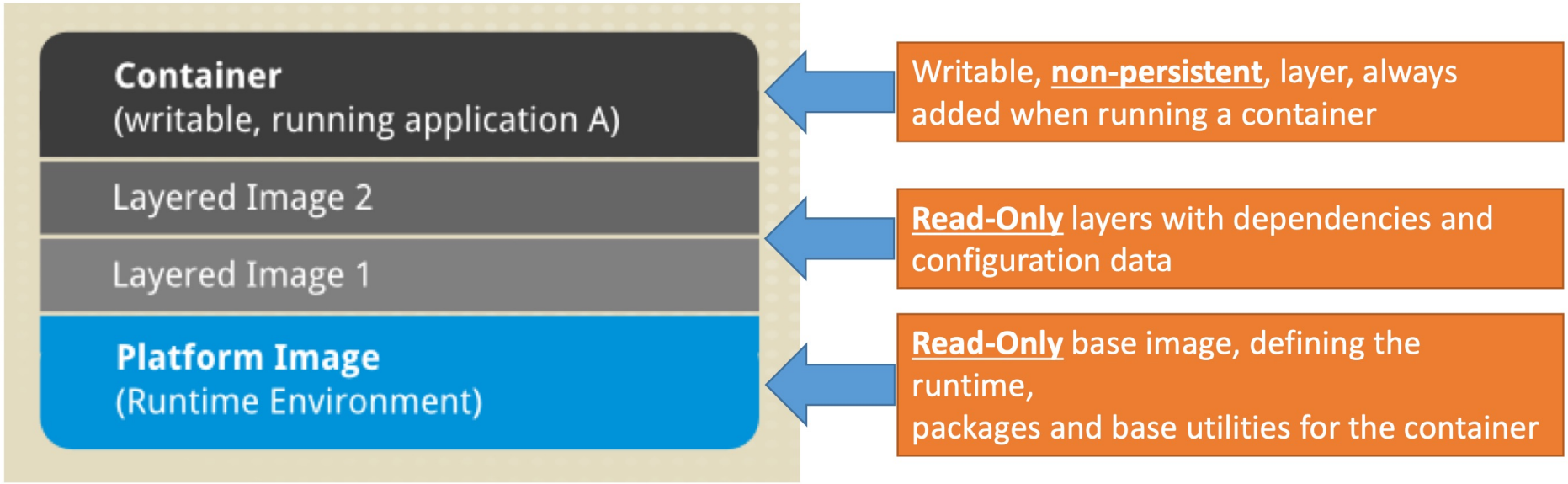
- Sequencia de comandos que preparam o container para execução
  - Lista de dependências
  - Lista de comandos a executar dentro do container
  - Conjunto de comandos a executar para iniciar container
- Usado localmente ou distribuído num registry
- Pode definir versões do mesmo software
  - Exemplo: nginx:latest, nginx:alpine

```
FROM debian:stretch-slim

LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

ENV NGINX_VERSION 1.15.9-1~stretch

ENV NJS_VERSION 1.15.9.0.2.8-1~stretch

RUN set -x \ && apt-get update \ && apt-get install --no-install-
recommends --no-install- suggests -y gnupg1 apt-transport-https ca-
certificates \ && \

...OMMITED...

RUN ln -sf /dev/stdout /var/log/nginx/access.log \ && ln -sf
/dev/stderr /var/log/nginx/error.log

EXPOSE 80

STOPSIGNAL SIGTERM

CMD ["nginx", "-g", "daemon off;"]
```

Imagens

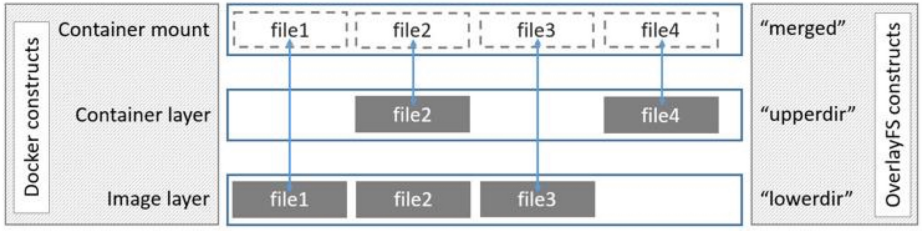
- Containers têm os seus dados iniciais em imagens
  - Dados, aplicação, bibliotecas
  - Contêm um *entrypoint* que é executado quando o container é iniciado
- Composto por múltiplas camadas
  - Imagem base
  - Várias imagens, alteram o conteúdo (com diferenças)
  - Usam *filesystems* por camadas (overlayfs, aufs, btrfs, ZFS)
- Read-only ou não persistentes
  - Servem de base para o container iniciar
  - Múltiplos containers podem partilhar a mesma imagem

Camadas de uma Imagem



Imagens

- É importante manter as imagens pequenas
  - Não basear imagem em distribuições completas (ex. Ubuntu), preferir distribuições próprias (ex. Alpine)
- Disponiveis no host
  - /var/lib/Docker/overlay2/<ID>/
    - /merged: filesystem visto dentro do container
    - /diff: diferenças para a base



Persistência Container

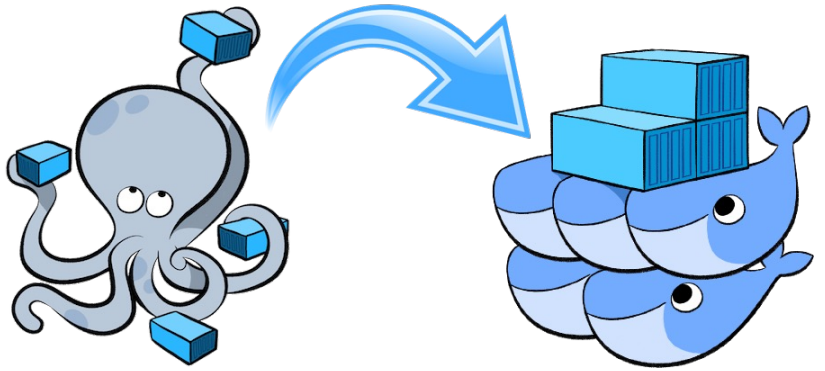
- Camada superior da imagem pode ser escrita, mas não persistida!
  - Parar o container implica perda total de dados
- Persistência é alcançada através de recursos externos
  - Bind mount: caminho no *host*
  - Volume: gerido como um recurso Docker
  - Externo: iSCSI, Database, NFS, etc

Container Network

- Containers executam com um configuração de rede especifica
  - Sem rede de *ingress* adicionada por omissão
- Mecanismos de rede típicos:
  - Bridge: *virtual switch*
  - Host: rede apenas entre *host* e container
  - None: completamente isolado
- Serviços dentro de containers não podem disponibilizar serviços a outras aplicações
  - Necessário expor explicitamente os serviços:

Importa ainda conhecer

- Docker Compose
  - Orquestração de containers Docker
- Kubernetes
  - Orquestrador (mais que containers)



kubernetes