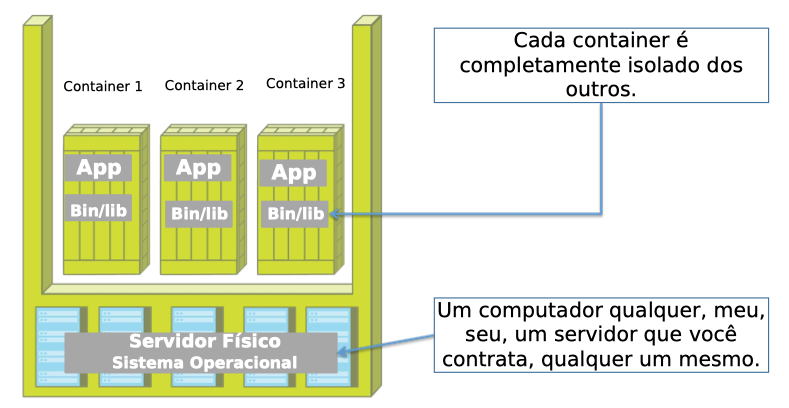
Resumo comandos Docker

O que são containers?

Containers, conhecidos também por Containers Linux, são uma maneira de criar ambientes isolados que podem executar código enquanto compartilham um único sistema operacional.



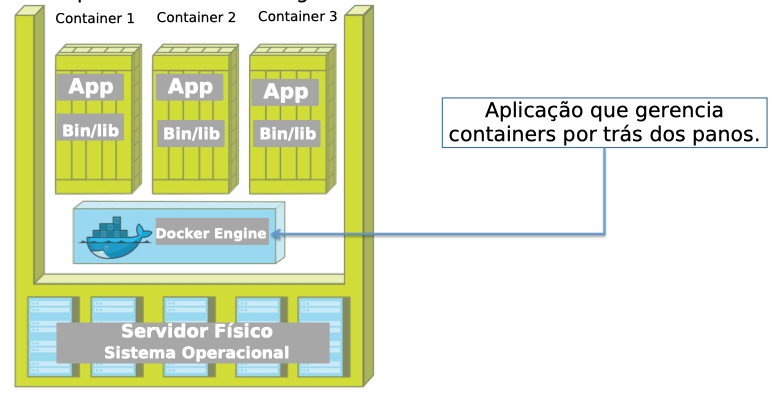
O que são Containers.

Ok, mas então, o que é o Docker?

Gerenciar containers é uma tarefa difícil.

O Docker é uma ferramenta que deixa a tarefa de gerenciar containers muito mais fácil.

Afinal, qual a diferença entre containers e máquinas virtuais?

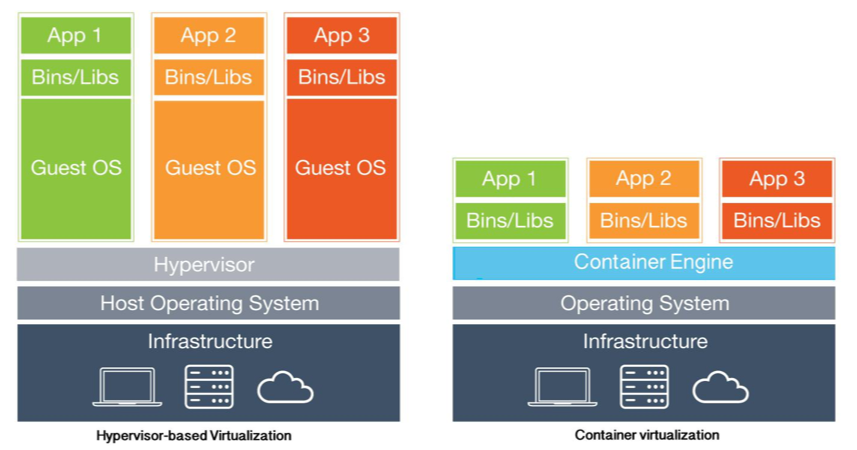


Docker - aplicação que facilita o processo de gerenciamento de containers.



Containers versus máquina virtual.

A diferença é bem significativa e muito favorável ao uso dos Containers, veja na figura a seguir as duas estruturas, maquinas virtuais e containers. Pode se dizer que Containers são uma evolução da Virtualização.



Máquinas virtuais e containers.

YAML - Yet Another Markup Language



YAML - linguagem de serialização de dados.

Yet Another Markup Language, criada em 2001 é uma linguagem não apenas de marcação, como o nome sugere, mas sim uma linguagem de **Serialização de Dados**.

**A linguagem YAML:**

- Se integra com outras linguagens, como Python, Ruby, Java, etc;

* + - * Possui tipos de dados comuns como escalares, listas, arrays, etc;
      * Comumente utilizada como arquivo de configuração ou armazenamento de dados;

**Tem como objetivos:**

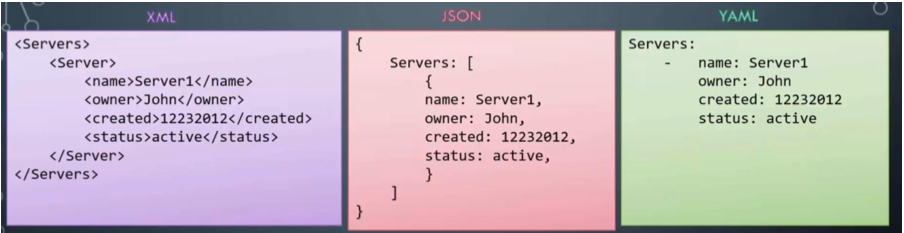
- Ser lida facilmente por humanos;

- Portátil;

- Integrar facilmente com outras linguagens;

- Fácil de implementar e usar;

Afinal para quê usamos a linguagem...



Porque usar YAML - observe como é simples e intuitiva.

YAML aceita as estrutura de dados mais comuns.

O que são imagens em um container?



Estruturas de dados comuns aceitas em YAML.

Quando falamos de containers, imagens são ‘**templates**’ (modelos) na qual usamos para criar containers.

Exemplo: Imagine uma aplicação que será desenvolvida com **Python** e banco de dados **PostgreSQL** e será executada no servidor web **Nginx**.

Podemos então criar 3 containers, sendo que:

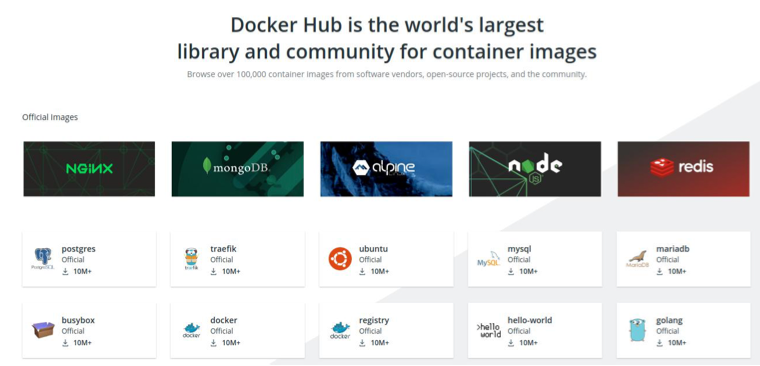
- Um container irá usar uma imagem com a linguagem Python contendo suporte a se comunicar com o banco de dados PostgreSQL;

- Um container irá usar uma imagem com o servidor de banco de dados PostgreSQL;

- Um container irá usar uma imagem com o servidor web Nginx;

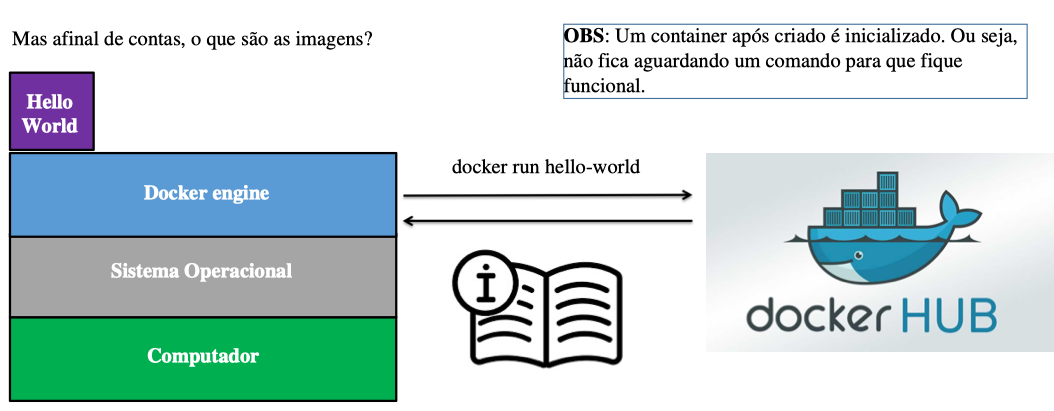
**Atenção**: Não confunda as imagens de containers com as imagens de serviço cloud como Amazon AWS, Google Cloud Plaftorm e Microsoft Azure.

**Nota:** primeiramente o docker tenta encontrar uma **imagem localmente** e caso não encontre ele faz o **download** da plataforma Docker Hub em: https://hub.docker.com/.



Plataforma Docker Hub.

**Obs.:** Uma imagem é criada a partir de um arquivo chamado **Dockerfile**. Podemos visualizar uma representação de o que é uma imagem na figura a seguir:

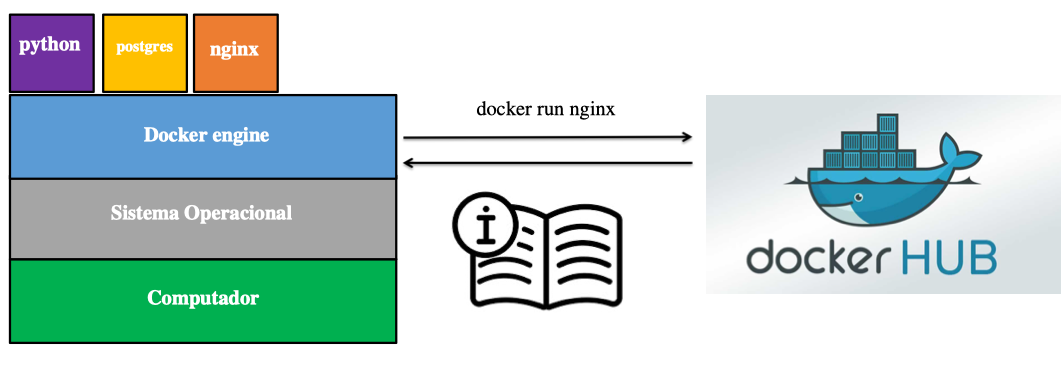


Visualização do que vem a ser uma imagem Docker.

**Importante**: Cada container é criado a partir de um arquivo Dockerfile (é um arquivo com instruções linha a linha para se criar uma imagem) que definirá uma.

**Obs.:** Quando solicitamos ao docker que crie um container baseado em alguma imagem esta imagem é procurada primeiro localmente e caso não encontrada o docker procura e caso encontre baixa do Docker Hub.

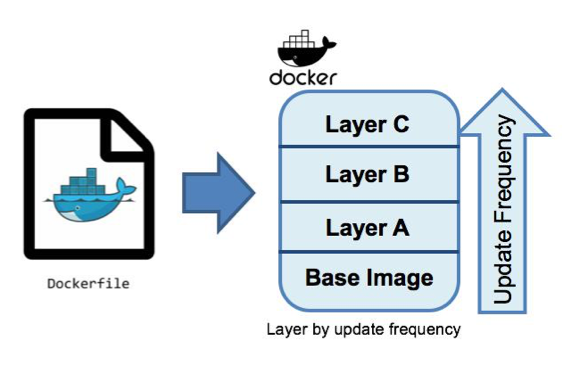
Após a criação de um container a partir de uma imagem esta imagem fica armazenada no nosso computador local, e caso resolvamos criar um outro container com esta mesma imagem não será necessário baixá-la novamente, a não ser que você tenha deletado a imagem.



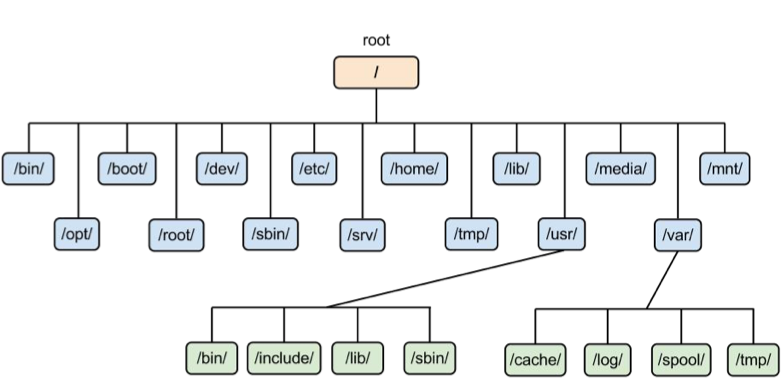
Estrutura final para a aplicação sugerida.

Sistemas de arquivos em camadas

Um dos tópicos mais importantes do Docker é o **filesystem**, ou seja, o sistema de arquivos utilizado pela ferramenta. O sistema de arquivos do Docker é chamado de **Layered**, ou seja, é um sistema de arquivos em camadas.



Sistema de arquivos em camadas.



Estrutura em camadas - árvore de diretórios - assim também é o filesystem

Mas o que isso quer dizer?

**Sistemas de Arquivos**

Um sistema de arquivos comum (Linux/Unix), possui basicamente duas camadas:

**bootfs**: Onde fica o sistema de Boot do sistema operacional e o Kernel.

**rootfs**: Inclui o sistema de arquivos do sistema operacional, incluindo a arquitetura de diretórios, como /dev, /proc, /bin, /etc, /lib, /usr e /tmp assim como os arquivos de configuração e binários do sistema operacional.

Quando o sistema operacional é iniciado ele carrega o **rootfs** primeiramente em modo somente leitura, verifica sua integridade e em seguida remonta-o como leitura/escrita e assim ficando disponível para o usuário e aplicações.

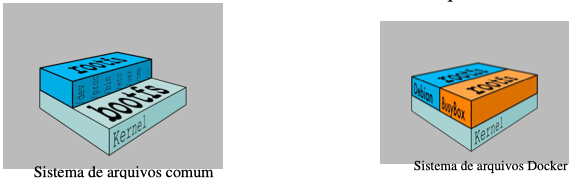
Sistemas de Arquivos no Docker

No **Docker** temos esta arquitetura também, mas com um pequeno diferencial:

A camada de escrita que o processo/aplicação visualiza **não** é o mesmo rootfs base do sistema mas **sim** uma camada de abstração do rootfs.

***Isso faz com que um container torne-se portável, pois as modificações realizadas não são aplicadas ao sistema origem do container e sim na camada a qual o sistema visualiza***.

Para termos um entendimento mais claro do funcionamento do sistema de arquivos:



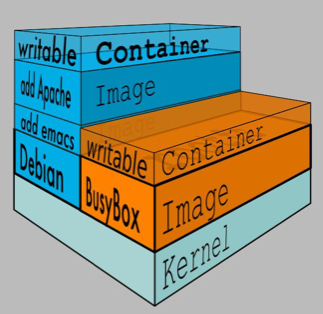
Diferenças no sistema de arquivos - comum e Docker.

Note que o **bootfs** em vez de ser único por **sistema/container** é compartilhado entre eles.

Já o **rootfs** *é isolado por camadas, ou seja, o que deve ser em comum entre o host e o container é compartilhado via AUFS\**, que monta uma camada de leitura/escrita em cima do sistema de arquivos que é somente leitura. Isso garante que modificações feitas dentro do container não afetem o sistema de arquivos do host.

Sistemas de Arquivos no Docker

Note um exemplo do funcionamento da arquitetura do sistema de arquivos do Docker:



Exemplo de um sistema de arquivos do Docker.

A cada modificação do container é gerada uma nova camada.

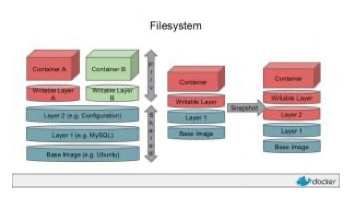
Por exemplo, imagine que iniciamos um container a partir de uma imagem do Debian, e então instalamos o Emacs no container. Então instalamos o servidor web Apache.

Se formos transformar este container em uma imagem, esta imagem possuirá a primeira camada que é do Debian e mais duas camadas adicionais, do Emacs e do Apache.

Sistema de Arquivos em Camadas

Imagens são compartilhadas entre containers Docker

Este tipo de sistema de arquivos em camadas busca sempre a eficiência. Desta forma, containers compartilham imagens semelhantes.



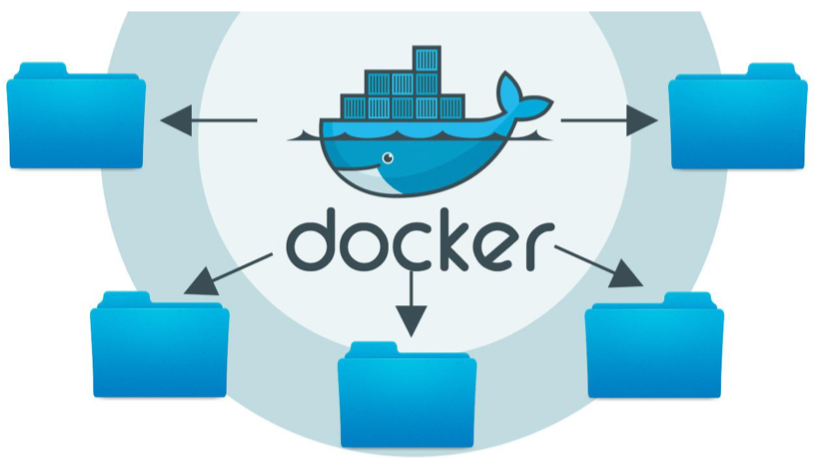
Exemplo de um sistema de arquivos em camadas.

Ou seja, imaginando que temos 2 containers, sendo que: No Container 1 instalamos o Debian, instalamos o Nginx e No Container 2 instalamos o Debian, instalamos o Redis

*A imagem do Debian será compartilhada entre os dois containers* (que é comum a ambos será compartilhada entre eles).

Os containers não persistem dados

Até então aprendemos a criar containers através de imagens no Docker Mas até o momento não tentamos persistir dados dentro de um container. Mas o que quero dizer com persistir dados?



Containers não persistem dados por default. É preciso habilitar essa opção.

Persistência de dados

Quando falamos em persistir dados estamos falando em fazer alterações (inserir, editar) dados no sistema de arquivos do container.

Por exemplo, o que ocorre se criarmos um container, criarmos um arquivo dentro deste container com um texto e então pararmos o container e iniciarmos o container novamente?

Se a imagem na qual o container foi criado não contém por padrão persistência de dados habilitada, o conteúdo criado será perdido.

Qual a solução para o problema de não persistência de dados nos containers?

A solução para persistência de dados em containers são os Volumes, também chamados de Volumes de Dados.

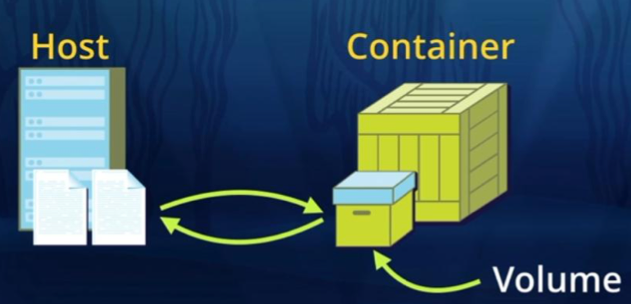
Aprenderemos sobre este recurso mais à frente!

Antes, faremos um exercício prático para checarmos este problema de não persistência nos containers.

O que são volumes em containers?

Dado que containers por padrão não persistem dados, temos que usar recursos para contornar esta situação. Este recurso são os **Volumes.**

*Fazendo uso de Volumes, “expomos” arquivos e diretórios da nossa máquina para o container*. Note que os arquivos/diretórios estarão persistidos no **Host** e **não** no **container**.



Volumes em containers.

Desta forma, quando paramos o container todos os dados presentes no volume exporto é preservado e quando iniciamos novamente o container os dados estão todos disponíveis.

Volumes são uma solução perfeita para nós, desenvolvedores, pois se antes, por exemplo, tínhamos que instalar um servidor web e um banco de dados diretamente na nossa máquina ou criar uma máquina virtual com estes servidores, agora podemos fazer uso de containers, por exemplo, um para servidor web e um para banco de dados, e termos os arquivos de desenvolvimento para trabalho local.

**OBS:** *Os arquivos/diretórios expostos estão na sua máquina (host) e não no container. Desta forma qualquer modificação neles a persistência é feita no seu computador e não no container*.



Persistência de dados com volumes em containers.

O que é um Dockerfile?

Nos comandos abaixo, vimos comandos para criar containers e também que esses comandos foram ficando maiores e mais complexos conforme um dos containers exemplificado abaixo:

Ainda após a criação do container para que possamos e ou queiramos instalar ferramentas, pacotes ou realizar configurações e este processo todo manual se torna lento, e susceptível a erros.

A tendência é realmente que usemos mais e mais parâmetros para informar ao Docker tudo que precisamos para os nossos container.

Isso dificulta a criação e manutenção de containers. Veja o comando usado abaixo, como exemplo.

* docker run —name servidor\_web -d -p 8080:80 -e NGINX\_ENTRYPOINT\_QUIET\_LOGS=1 -v /Users/accol/downloads/persistir:/usr/share/nginx/html nginx:1.23.3-alpine.

Para resolver este tipo de problema usamos o Dockerfile.

Um Dockerfile nada mais é do que um documento texto, no formato YAML, com pequenas instruções que são usadas pelo Docker para gerar uma imagem baseada nas instruções do documento. Com esta imagem criamos nosso container.



Dockerfile define a forma como a imagem será executada no container.

Esta imagem gerada a partir do nosso Dockerfile serve também para que publiquemos na nossa conta no Docker Hub e desta forma possamos distribuir esta imagem facilmente para outras pessoas.

Como funciona a comunicação em uma rede Docker?

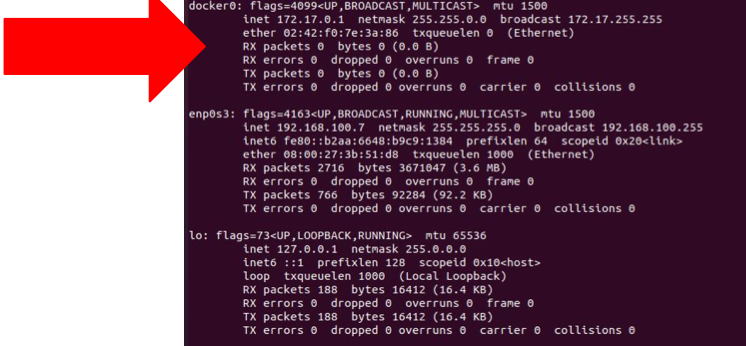
Computadores se comunicam em uma rede através de endereços IP (Internet Protocol). Os endereços IP servem de endereçamento do dispositivo e de identificador do dispositivo na rede. Não pode haver repetição de um endereço IP em uma rede.

Quando trabalhamos com Docker este cria uma rede interna, em outra classe, e distribui IPs aos containers criados.



Exemplo de uma estrutura de computadores em uma Rede local.

Após a criação de sua estrutura em redes, use o comando ifconfig para visualizar a rede criada pelo Docker.



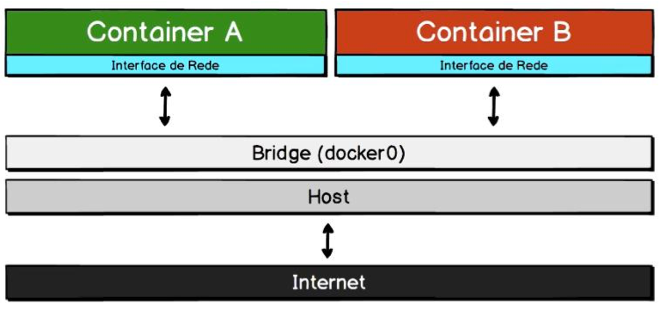
Exemplo do retorno do comando ifconfig.

Quando estamos trabalhando com vários containers em um projeto, devemos levar em conta, por motivos de segurança, quais containers precisam/podem ter seus serviços/IPs/portas expostas e quais não.

Por exemplo, um serviço HTTP precisa ter a porta exposta para que as pessoas acessem a aplicação web.

Mas um banco de dados que será acessado apenas pela aplicação web e não pelas pessoas fora do contexto do Docker não deveria ter seu IP/porta exposta.

O Docker por padrão utiliza um “driver” de rede conhecido como Bridge



Estrutura em camadas de uma rede criada pelo Docker.

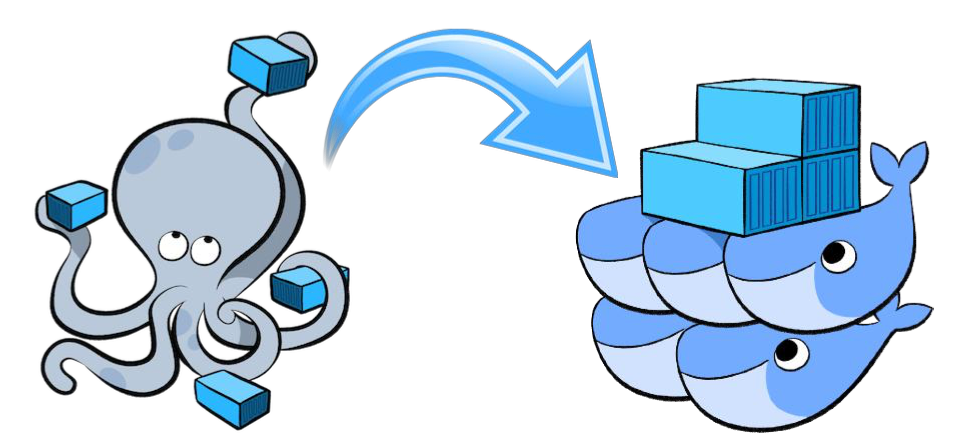
Desta forma, ao criarmos um container, este irá receber uma interface de rede própria e um endereçamento IP. O Docker fará então uma “ponte” entre o container e o host (nosso computador).

Outros tipos de rede no Docker

Além do modelo padrão, “bridge” podemos ter no Docker os seguintes tipos:

* + None Network, no qual o próprio nome diz que não haverá nenhuma rede;
  + Host Network, na qual ao invés de termos a “ponte” entre os containers e o Host (nosso computador) o acesso é direto entre o container e o host;
  + Overlay Network, usado com o Docker Swarm que é um orquestrador\* de containers;

O que é o Docker Compose?



Ícone do Docker Compose.

Até este momento, foi possível perceber de forma bastante clara o problema que temos quando no nosso projeto temos mais de 1 container.

Claro que nada nos impede de criar 1 container apenas, instalar e configurar tudo nele e colocar nossa aplicação para “rodar”, mas isso por mais que pareça simples e fácil trará muitos problemas, começando que sua aplicação não irá escalar facilmente já que não conseguimos escalar os **serviços** de forma individual.

Outro ponto importante é que ficar lembrando e executando os comandos e parâmetros do docker não é tão prático, pois uma digitação errada poderá comprometer a aplicação.

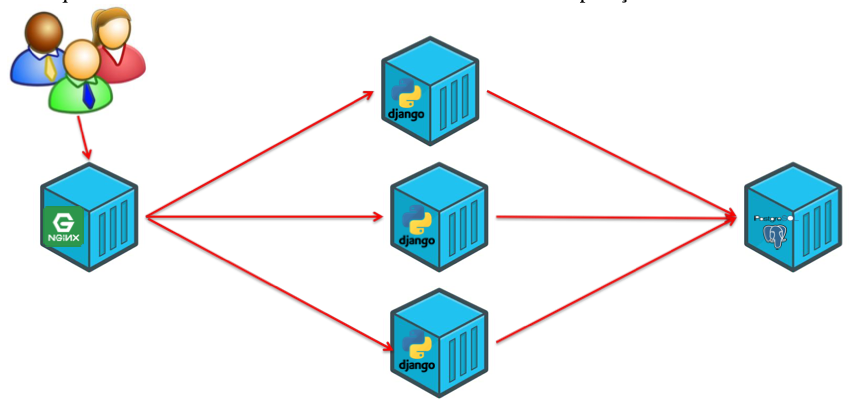
docker build -f pgfusion.dockerfile -t accol/pgfusion:v1

docker build -f fusion.dockerfile -t accol/fusion:v1

docker run —name pgfusion -d accol/pgfusion:v1

docker run —name pgfusion -d -p 8080:8000 accol/fusion:v1

Imagine ter que executar os comandos do docker manualmente tendo uma aplicação real?



Simulação de um ambiente de desenvolvimento, bastante simples, mas não deixa de ser complexo.

Por outro lado, aprendemos a criar o Dockerfile no qual podemos escrever o arquivo de definição do container e após isso “buildar” a imagem.

Isso já facilita pois podemos checar os comandos antes da execução e teremos o arquivo que poderemos reutilizar e controlar a versão com **git** juntamente com o restante do nosso código.

Mas lembre-se que o **Dockerfile** serve apenas de definição do container para que a imagem do mesmo seja gerada. Ou seja, *no final das contas ainda temos que ficar digitando comandos docker com todos os parâmetros necessários*.

Mas felizmente temos o Docker Compose que dado tudo que aprendemos até então pode nos ajudar muito na nossa jornada.

O Docker Compose nada mais é do que um arquivo **YAML** que concentra todos os comandos e configurações para que os containers sejam criados.

***Ou seja, ao invés de ficarmos digitando os comandos manualmente no terminal, criamos os Dockerfiles, criamos o Docker Compose descrevendo tudo que precisamos para ter os containers e executamos o Docker Compose que irá ler o arquivo e fazer todo o trabalho sujo pra gente***.

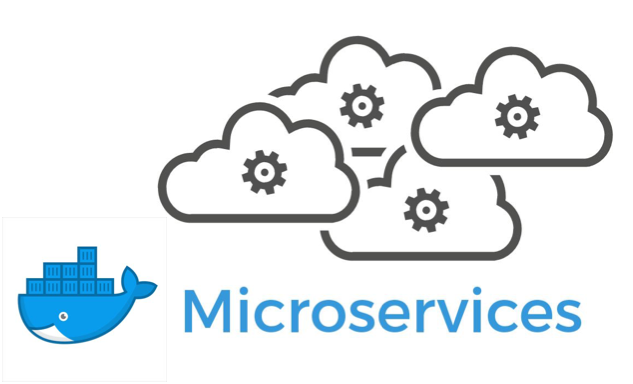
Instalação do Docker Compose

Em versões anteriores do Docker o Docker Compose fazia parte da instalação padrão mas atualmente é uma aplicação independente.

Isso porque o Kubernetes assumiu o lugar da Orquestração de Containers e faz isso muito melhor.

Então você precisará instalar seguindo este tutorial (<https://docs.docker.com/compose/install/>) para que possamos usá-lo nas próximas aulas.

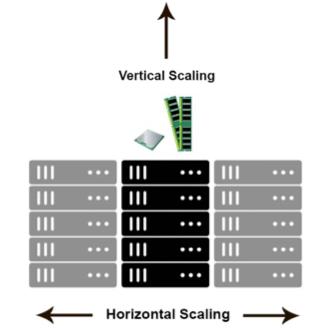
Docker e Microserviços



Docker e microserviços.

Escalabilidade

* + Escalabilidade vertical: É quanto temos que aumentar a capacidade da máquina que estamos utilizando, por exemplo, aumentar a memória RAM ou espaço no HD.
  + Escalabilidade horizontal: É quanto replicamos a mesma máquina ou serviço N vezes.

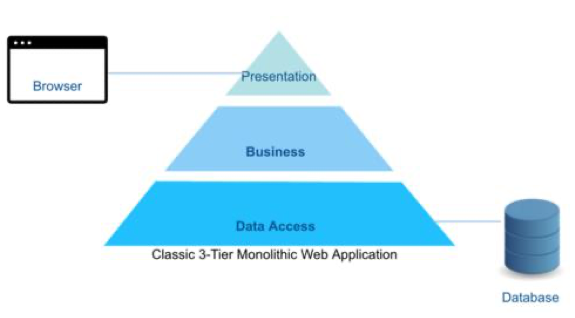


Escalabilidade horizontal e vertical

**Aplicação Monolítica**

Uma aplicação monolítica é aquela na qual toda a base de código está contida em um só lugar, ou seja, todas as funcionalidade estão definidas no mesmo bloco.

Geralmente este bloco é dividido em 3 partes:



Aplicação Monolítica.

A camada de Apresentação é a camada de visualização que será mostrada para o usuário. Por exemplo, no caso de uma aplicação web, esta camada contém as páginas HTML com JS e CSS que serão renderizadas pelo browser do cliente que a acessar.

A camada de Negócio é a camada que contém a lógica da aplicação. Nesta camada geralmente se encontram todas as bases de código, chamadas, APIs e literalmente toda a inteligência do sistema.

Na camada de dados temos apenas as classes responsáveis pela conexão com o sistema de armazenamento de dados.

Docker e Microserviços

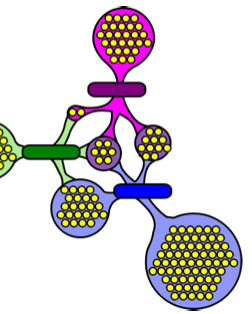
**Vantagens de Aplicações Monolíticas**

* + Simplicidade da arquitetura, pois não existem muitas camadas para se preocupar;
  + Agregação de tecnologia, pois toda aplicação é desenvolvida em uma mesma tecnologia, facilitando a coesão da equipe;
  + Fluxo de publicação simples, pois alterou? Compilou? É só publicar!;
  + Rápido desenvolvimento, pois já que usa uma arquitetura mais simples, o seu desenvolvimento tende a ser muito mais rápido;

Aplicações Microserviços

Com uma crescente utilização de APIs e rotas cada vez mais genéticas o termo escalabilidade ganhou força de forma que seria possível escalar qualquer serviço através da escala horizontal.

As empresa começaram então a adotar uma prática pouco comum no ambiente de desenvolvimento: Quebrar a lógica de negócio em minúsculos pedaços independentes que se completam, criando uma espécie de rede de APIs internas totalmente ou parcialmente conectadas.

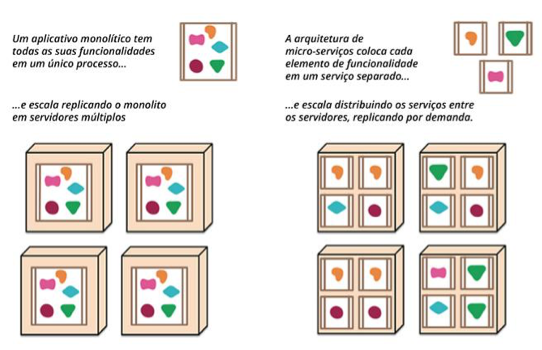


Arquitetura microserviços e integração via APIs.

Uma das vantagens desta arquitetura é que agora é possível escalar qualquer tipo de serviço individualmente sem a necessidade de escalar o ambiente todo como era feito antes na arquitetura monolítica.

Mas o que é um Microserviço?

Segundo **Martin Fowler**, microserviço é uma abordagem que desenvolve um aplicativo único como uma suite de pequenos serviços.



Microserviços e sua escalabilidade.

Aplicações Microserviços

Apesar da ideia de microserviços ser muito simples ao mesmo tempo é muito complicado de se desenvolver de forma sustentável pois ao invés de lidar com um único “monstro” agora vamos dividi-lo em minúsculos “monstrinhos”.

Em geral, um microserviço é um sistema simples, geralmente uma API, que se comunica através de HTTP.

Uma característica importante dos microserviços é que eles podem, individualmente, serem desenvolvidos em uma linguagem de programação diferente, utilizando diferentes tecnologias de persistência de dados.

Vantagens de Aplicações Microserviços

* + - Arquitetura individual simples;
    - Mecanismo de comunicação universal e leve (HTTP);
    - Sistemas totalmente independentes;
    - Facilidade de deploy e testes unitários;
    - Ausência de um ponto de falha único;

Desvantagens de Aplicações Microserviços

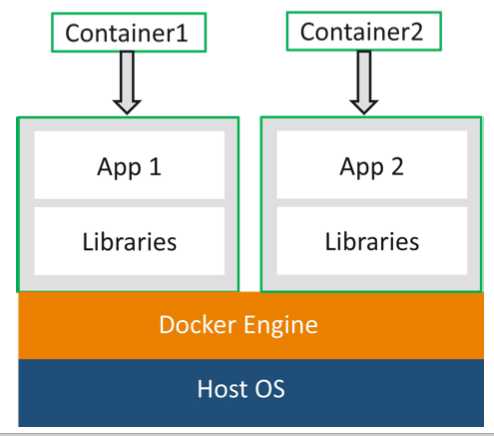
* Se não for bem planejado e bem executado pode se transformar em uma grande bagunça;
* A arquitetura geral pode se tornar complexa se não for bem documentada;



Docker na nuvem.

O que é o Docker Cloud?

É preciso tem sempre em mente o que é o Docker, sendo assim, relembremos, note que o Docker Engine, torna o Docker independente da Plataforma:



Docker Engine o segredo por trás do Docker.

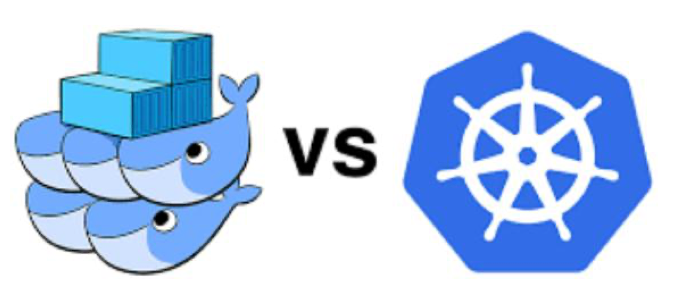
Outro ponto relevante é que o ambiente de desenvolvimento é o mesmo da produção, sendo assim, nada te impede de contratar um serviço de Virtual Private Cloud e fazer uso do Docker.

**Docker vs Kubernetes**

Docker é uma empresa, criadora da ferramenta Docker Engine que serve para facilitar a criação e gerenciamento de containers.

A empresa Docker tem outra ferramenta, chamada Docker Swarm que pode ser utilizada para orquestração de containers.

O Kubernetes é uma ferramenta criada pela Google que virou padrão do mercado para orquestração de containers.



Docker Swarm versus Kubernetes - gerenciador de containers padrão de mercado.

Alguns comandos interessantes

* docker version - exibe a versão detalhada do Docker que está rodando.
* docker —version - exibe apenas a versão do Docker.
* docker —help - traz uma lista de todos os comandos do Docker.
* docker container ls - lista todos os containers ativos.
* docker container —help - lista todos os comandos Docker relativos a containers.
* docker ps - lista todos os containers em execução.
* docker ps -a - lista todos os containers em execução e parados.
* docker container start <id\_container> - inicializa o container <id\_container>.
* docker container stop <id\_container> - para a execução de um container com o <id\_container>.
* docker pull <nome\_da\_imagem> - comando apenas baixa o <nome\_da\_iamgem>, ou seja, não coloca o container em execução. Note que o não especificar a versão da imagem recairá na opção **latest** que é o **default**, recomenda-se **nunca usar a opção default**, mas sim, especificar o número da versão desejada (a versão na qual sua aplicação foi construída).
* docker pull <nome\_da\_imagem>:<número\_da\_versão\_desejada> - baixa a imagem de nome especificada por <nome\_imagem> na versão especificada <número\_da\_versão\_desejada>.
* docker run hello-world - uma imagem docker para Hello word
* docker run —name <nome\_imagem> -d <imagem\_desejada>:<v1> - baixa e executa o container com o imagem de nome <nome\_imagem>.f
* docker image ls - lista todas as imagens presentes em sua máquina.
* docker image rm <id\_imagem> - deleta a imagem com o identificador especificado por <id\_imagem>.
* docker image rm <id\_imagem> -f - força um delete se existir um container, este será parado e deletado para que a imagem possa ser deletada. No caso de haver uma cópia da imagem, será preciso usar o -f, caso não o faça, não será possível deletar a imagem fonte da cópia. Atenção, tanto a cópia quanto a imagem serão deletados.
* docker image inspect <id\_imagem> - faz a inspeção na imagem baixada.
* docker image tag <nome\_da\_imagem>:<versão\_da\_imagem> <nome\_novo> - faz uma cópia da imagem para <nome\_novo>.
* docker run <nome\_da\_imagem> - este comando cria um container a partir de uma imagem baixada com o comando - docker pull <nome\_da\_imagem>.
* docker run -d -p 8080:80 <imagem> - configura a porta 80 da máquina local com a porta 80 da <imagem> do Docker, por exemplo, para acessar algum servidor web (nginx por exemplo). **Atenção:** o acesso é via **http** e **não** via **https**.
* docker run —name linux\_alpine —rm -I -t alpine:3.17.2 - este comando cria um container Docker rodando o Linux\_alpine na versão 3.17.2 e apresenta o shell.
* CTRL + D - sai de um container e devolve o terminal de sua máquina.
* docker run —name servidor\_web -d -p 8080:80 -e NGINX\_ENTRYPOINT\_QUIET\_LOGS=1 -v /Users/accol/downloads/persistir:/usr/share/nginx/html nginx:1.23.3-alpine - mapeia um diretório local <Users/accol/downloads/persistir> no container nginx em </usr/share/nginx/html>. Este é um exemplo básico de criação de Volumes em Containers.
* docker exec -it <nome\_container> sh - este comando executa de forma iterativa o shell de um container <nome\_container> exibindo o shell.
* docker inspect <id\_container> - sabemos que trará muitas informações a cerca do container, para este caso - montagem - temos uma descrição que nos mostra o mapeamento exato que foi criado
* docker build -f <nome\_arq\_dockerfile> -t <nome\_usuário>/<nome\_serviço>:<versão> . - este comando cria a partir de um arquivo dockerfile um container e gera uma tag (-t) com o nome do usuário no Docker Hub para que possa ser publicada. Note o ponto no final, ele diz onde está o arquivo dockerfile (utilizado para gerar o build - no caso o ponto significa diretório atual).
* docker login —username=<seu\_username> - permite que você log na sua conta no docker hub.
* docker image push <seu\_usuário>/<sua\_imagem>:<tag> - publica sua imagem no docker hub.
* docker network ls - lista as redes dos tipos do Docker, note que, cada tipo de rede tem seu identificador próprio - seu id.
* docker run —net <tipo\_rede> <Nome\_imagem> <comando\_a\_ser\_executado> - define o tipo de rede que o Docker deverá aplicar a seu container <tipo\_rede> que pode ser: default - não precisa digitar nada, none, host e overlay.
* docker inspect <tipo\_rede> - permite inspecionar o tipo da rede.
* docker network create —driver <tipo\_rede (bridge)> <nome\_faixa> - este comando permite que se crie uma outra faixa de endereços ip para seus containers.
* docker network connect bridge <nome\_da\_faixa\_desejada> - este é um exemplo de comando docker para criar uma ponte de comunicação entre duas faixas de IP diferentes. Atenção que a ponte criada é unidirecional.
* docker network disconnect bridge <nome\_da\_faixa\_desejada> - este é um exemplo de comando docker para desconectar uma ponte de comunicação entre duas faixas de IP diferentes.
* docker-compose build - executa o arquivo docker-compose preparando toda aplicação. Nota, este comando "builda" as imagens da aplicação.
* docker-compose up -d - executa o docker compose, a diretiva -d evita que o terminal fique preso na aplicação. Neste momento são criados os containers, redes e volumes.
* docker-compose up - executa como antes sem a opção d temos acesso aos logs da execução, que podem ser úteis para identificação e remoção de erros.
* docker-compose down - para os serviços em execução.