Curso de Docker

para Desenvolvedores

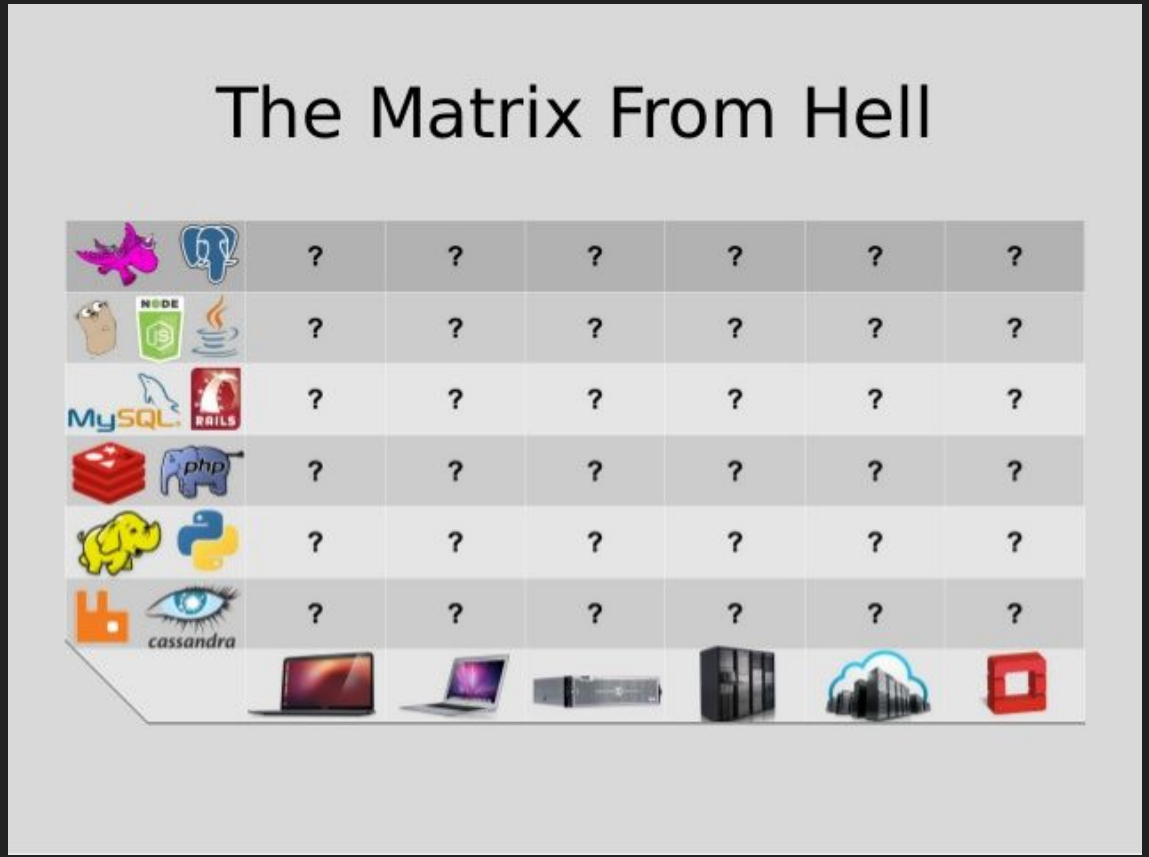
O que é Docker?

- O **Docker** é um software que **reduz a complexidade de setup** de aplicações;
- Onde **configuramos containers**, que são como servidores para rodar nossas aplicações;
- Com facilidade, podemos criar **ambientes independentes** e que funcionam em diversos SO’s;
- E ainda deixa os projetos **performáticos**;

Por quê Docker?

- O **Docker** proporciona mais velocidade na configuração do ambiente de um dev;
- **Pouco tempo gasto em manutenção**, containers são executados como configurados;
- **Performance** para executar aplicação, mais performático que uma VM;
- Nos livra da **Matrix from Hell**;

Conhecendo a Matrix from Hell



Qual versão utilizar?

- O **Docker** é dividido em duas versões: **CE x EE**
- CE é a **Community Edition**, edição gratuita, que nos possibilita utilizar o Docker normalmente, é a que vamos optar;
- EE é a **Enterprise Edition**, edição paga, há uma garantia maior das versões que são disponibilizadas e você tem suporte do time do Docker;

Instalação Windows

- Para Windows vamos instalar um software chamado **Docker Desktop**;
- Com ele virá a possibilidade também de rodar **Docker no terminal**, que é onde aplicaremos a maioria dos comandos durante o curso;
- Docker Desktop é uma **interface** para trabalhar com o Docker;
- Obs: Há duas versões, a que você vai instalar **depende da versão do seu Windows**;

Instalação Mac

- Para Mac vamos instalar um software chamado **Docker Desktop**;
- Com ele virá a possibilidade também de rodar **Docker no terminal**, que é onde aplicaremos a maioria dos comandos durante o curso;
- Docker Desktop é uma **interface** para trabalhar com o Docker;
- Este software é o mesmo que utilizamos no Windows;

Instalação Linux

- Para **Linux** vamos precisar escolher a versão baseada em nossa distribuição;
- Vamos precisar **executar comandos no terminal**, seguindo a documentação;
- Desta maneira teremos o Docker disponível;

Problemas de instalação

- Caso você ainda não conseguiu instalar, por algum erro ou incompatibilidade visite o site: https://docs.docker.com/get-docker/
- Nele há um guia para **cada sistema operacional** e também por distribuição de Linux;
- Infelizmente ainda não há uma maneira global e única de instalar o Docker;

Editor de código do curso

- Neste curso vamos utilizar o **VS Code**;
- Ele possui um **terminal integrado**, o que ajuda muito a trabalhar com o Docker;
- Além de ser, provavelmente, o **mais utilizado** em empresas atualmente;
- Obs: é opcional;

Extensão Docker do VS Code

- A **extensão de Docker** no VS Code vai ajudar a criar código para arquivos de Docker no editor;
- Esta extensão **sugere código (auto complete)** e também trás um **highlight** que ajuda a programar arquivos Docker;
- Vamos **escrever nossas imagens** com ajuda dela e do VS Code;

Terminal no Windows

- Caso você opte **por não utilizar o VS Code**, vai precisar de uma forma de rodar comandos no terminal;
- A opção mais interessante é o **Cmder**;
- Este software simula um terminal de Linux, aceitando todos os comandos compatíveis com o Docker também;

Testando o Docker!

- Vamos testar o Docker utilizando uma **imagem real**;
- Para rodar containers utilizamos o comando **docker run**;
- Neste comando **podemos passar diversos parâmetros**;
- Neste exemplo vamos passar apenas o nome da imagem que é **docker/whalesay**
- Um comando chamado cowsay e uma mensagem;

Curso de Docker

Conclusão da seção

Containers

Introdução da seção

O que são containers?

- Um **pacote de código que pode executar uma ação**, por exemplo: rodar uma aplicação de Node.js, PHP, Python e etc;
- Ou seja, os nossos projetos serão executados dentro dos containers que criarmos/utilizarmos;
- **Containers utilizam imagens** para poderem ser executados;
- **Múltiplos containers podem rodar juntos**, exemplo: um para PHP e outro para MySQL;

Container x Imagem

- **Imagem e Container** são recursos fundamentais do Docker;
- Imagem é o **“projeto”** que será executado pelo container, todas as instruções estarão declaradas nela;
- Container é o **Docker rodando alguma imagem**, consequentemente executando algum código proposto por ela;
- O fluxo é: programamos uma imagem e a executamos por meio de um container;

Onde encontrar imagens?

- Vamos encontrar imagens no repositório do Docker: **https://hub.docker.com**
- Neste site podemos **verificar quais as imagens existem** da tecnologia que estamos procurando, por exemplo: Node.js;
- E também **aprender a como utilizá-las**;
- Vamos executar uma imagem em um container com o comando: **docker run <imagem>**

Verificar containers executados

- O comando **docker ps ou docker container ls** exibe quais containers estão sendo executados no momento;
- Utilizando a **flag -a**, temos também todos os containers já executados na máquina;
- Este comando é útil para **entender o que está sendo executado e acontece** no nosso ambiente;

Executar container com interação

- Podemos rodar um container e deixá-lo **executando no terminal**;
- Vamos utilizar a **flag -it**;
- Desta maneira **podemos executar comandos disponíveis no container** que estamos utilizando o comando run;
- Podemos utilizar a imagem do ubuntu para isso!

Container X VM (Virtual Machine)

- **Container é uma aplicação que serve para um determinado fim**, não possui sistema operacional, seu tamanho é de alguns mbs;
- VM possui sistema operacional próprio, tamanho de gbs, **pode executar diversas funções ao mesmo tempo**;
- Containers acabam gastando menos recursos para serem executados, por causa do seu uso específico;
- VMs gastam mais recursos, porém podem exercer mais funções;

Executar container em background

- Quando iniciamos um container que persiste, **ele fica ocupando o terminal**;
- Podemos executar um container em background, para não precisar ficar com diversas abas de terminal aberto, utilizamos a **flag -d** (detached);
- Verificamos **containers em background com docker ps** também;
- Podemos utilizar o nginx para este exemplo!

Expor portas

- Os **containers de docker não tem conexão com nada de fora deles**;
- Por isso precisamos expor portas, a **flag é a -p** e podemos fazer assim: -p 80:80;
- Desta maneira **o container estará acessível na porta 80**;
- Podemos testar este exemplo com o nginx!

Parando containers

- Podemos parar um container com o comando **docker stop <id ou nome>**;
- Desta maneira estaremos liberando recursos que estão sendo gastos pelo mesmo;
- Podemos verificar os containers rodando com o comando **docker ps**;

Hack para execução de comandos

- Podemos utilizar qualquer comando que necessite um id de um container ou imagem com **apenas seus 3 primeiros dígitos**;
- docker stop a2b;
- **O Docker é inteligente o suficiente** para entender essa abreviação;
- E ele tenta ao máximo **criar ids únicos**;

Iniciando container

- Aprendemos já a parar um container com o stop, para voltar a rodar um container podemos usar o comando **docker start <id>**;
- Lembre-se que **o run sempre cria um novo container**;
- Então caso seja necessário aproveitar um antigo, opte pelo start;

Definindo nome do container

- Podemos definir um nome do container com a flag **--name**;
- Se não colocamos, **recebemos um nome aleatório**, o que pode ser um problema para uma aplicação profissional;
- A flag run é inserida junto do **comando run**;

Verificando os logs

- Podemos **verificar o que aconteceu em um container** com o comando logs;
- Utilizamos da seguinte maneira: **docker logs <id>**
- As últimas ações realizadas no container, serão **exibidas no terminal**;

Removendo containers

- Podemos **remover um container da máquina** que estamos executando o Docker;
- O comando é **docker -rm <id>**;
- Se o container estiver rodando ainda, podemos utilizar a **flag -f** (force);
- O container removido não é mais listado em docker ps -a;

Containers

Conclusão da seção

Containers e imagens

Introdução da seção

O que são imagens?

- Imagens **são originadas de arquivos que programamos** para que o Docker crie uma estrutura que execute determinadas ações em containers;
- Elas contém informações como: imagens base, diretório base, comandos a serem executados, porta da aplicação e etc;
- Ao rodar um container baseado na imagem, **as instruções serão executadas em camadas**;

Como escolher uma boa imagem

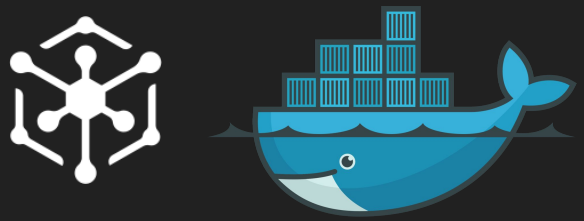
- Podemos fazer download das imagens em: https://hub.docker.com;
- Porém **qualquer um pode fazer upload de uma imagem**, isso é um problema;
- Devemos então nos atentar as **imagens oficiais**;
- Outro parâmetro interessante é a **quantidade de downloads** e a **quantidade de stars**;

Criando uma imagem

- Para criar uma imagem vamos precisar de um arquivo **Dockerfile** em uma pasta que ficará o projeto;
- Este arquivo vai precisar de algumas instruções para poder ser executado;
- **FROM**: imagem base;
- **WORKDIR**: diretório da aplicação;
- **EXPOSE**: porta da aplicação;
- **COPY**: quais arquivos precisam ser copiados;

Executando uma imagem

- Para executar a imagem primeiramente **vamos precisar fazer o build**;
- O comando é **docker build <diretório da imagem>**;
- Depois vamos utilizar o **docker run <imagem>** para executá-la;



Alterando uma imagem

- Sempre que alterarmos o código de uma imagem **vamos precisar fazer o build novamente**;
- Para o Docker é como se fosse **uma imagem completamente nova**;
- Após fazer o build vamos executá-la por o outro id único criada com o docker run;

Camadas das imagens

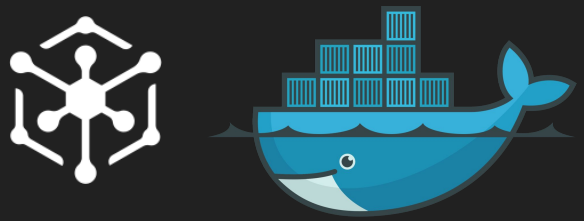
- As imagens do Docker são divididas em **camadas** (layers);
- Cada instrução no Dockerfile **representa uma layer**;
- Quando algo é atualizado **apenas as layers depois da linha atualizada são refeitas**;
- O resto permanece em cache, tornando o **build mais rápido**;

Download de imagens

- Podemos **fazer o download de alguma imagem** do hub e deixá-la disponível em nosso ambiente;
- Vamos utilizar o comando **docker pull <imagem>**;
- Desta maneira, caso se use em outro container, **a imagem já estará pronta para ser utilizada**;

Aprender mais sobre os comandos

- Todo comando no docker tem acesso a uma **flag --help**;
- Utilizando desta maneira, **podemos ver todas as opções disponíveis nos comandos**;
- Para relembrar algo ou executar uma tarefa diferente com o mesmo;
- Ex: **docker run --help**;



Múltiplas aplicações, mesmo container

- Podemos inicializar **vários containers com a mesma imagem**;
- As aplicações funcionarão em paralelo;
- Para testar isso, podemos determinar uma **porta diferente** para cada uma, e rodar no **modo detached;**

Alterando o nome da imagem e tag

- Podemos **nomear a imagem** que criamos;
- Vamos utilizar o comando **docker tag <nome>** para isso;
- Também podemos **modificar a tag**, que seria como uma versão da imagem, semelhante ao git;
- Para inserir a tag utilizamos: **docker tag <nome>:<tag>**

Iniciando imagem com um nome

- Podemos **nomear a imagem já na sua criação**;
- Vamos utilizar a **flag -t**;
- É possível inserir o nome e a tag, na sintaxe: **nome:tag**
- Isso torna o processo de nomeação mais simples;

Comando start interativo

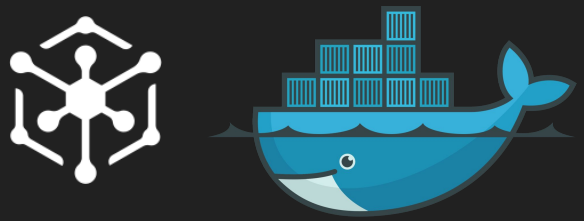
- A **flag -it** pode ser utilizada com o comando start também;
- Ou seja, não precisamos criar um novo container para utilizá-lo no terminal;
- O comando é: **docker start -it <container>**

Removendo imagens

- Assim como nos containers, **podemos remover imagens com um comando**;
- Ele é o: **docker rmi <imagem>**
- Imagens que estão sendo utilizadas por um container, apresentarão um erro no terminal;
- Podemos utilizar a **flag -f** para forçar a remoção;

Removendo imagens e containers

- Com o comando **docker system prune**;
- Podemos **remover imagens, containers e networks** não utilizados;
- O sistema irá exigir uma confirmação para realizar a remoção;



Removendo container após utilizar

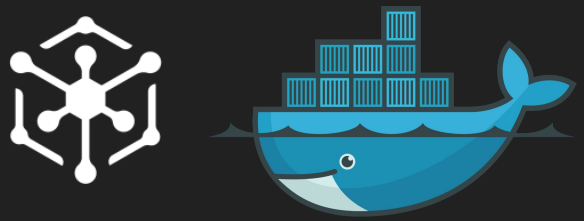
- Um container pode ser automaticamente deletado após sua utilização;
- Para isso vamos utilizar a **flag --rm**;
- O comando seria: **docker run --rm <container>**;
- Desta maneira **economizamos espaço no computador** e deixamos o ambiente mais organizado;

Copiando arquivos entre containers

- Para cópia de arquivos entre containers utilizamos o comando: **docker cp**
- Pode ser utilizado para copiar um arquivo de um diretório para um container;
- Ou de um container para um diretório determinado;

Verificar informações de processamento

- Para verificar dados de execução de um container utilizamos: **docker top <container>**
- Desta maneira temos acesso a quando ele foi iniciado, id do processo, descrição do comando CMD;



Verificar dados de um container

- Para verificar diversas informações como: **id, data de criação, imagem e muito mais**;
- Utilizamos o comando **docker inspect <container>**
- Desta maneira conseguimos entender como o container está configurado;

Verificar processamento

- Para verificar os processos que estão sendo executados em um container, utilizamos o comando: **docker stats**
- Desta maneira temos acesso ao andamento do processamento e memória gasta pelo mesmo;

Autenticação no Docker Hub

- Para concluir esta aula vamos precisar criar uma conta no: https://hub.docker.com
- Para autenticar-se pelo terminal vamos utilizar o comando **docker login**;
- E então inserir usuário e senha;
- Agora podemos **enviar nossas próprias imagens** para o HUB! =)

Logout do Docker Hub

- Para remover a conexão entre nossa máquina e o Docker Hub, vamos utilizar o comando **docker logout**;
- Agora não podemos mais enviar imagens, pois não estamos autenticados;

Enviando imagem para o Docker Hub

- Para enviar uma imagem nossa ao Docker Hub utilizamos o comando **docker push <imagem>**;
- Porém antes vamos precisar **criar o repositório** para a mesma no site do Hub;
- Também será necessário **estar autenticado**;

Enviando atualização de imagem

- Para enviar uma atualização **vamos primeiramente fazer o build**;
- **Trocando a tag da imagem** para a versão atualizada;
- **Depois vamos fazer um push** novamente para o repositório;
- Assim todas as versões estarão disponíveis para serem utilizadas;

Baixando e utilizando a imagem

- Para baixar a imagem podemos utilizar o comando **docker pull <imagem>**
- E depois criar um novo container com **docker run <imagem>**
- E pronto! Estaremos utilizando a nossa imagem com um container;

Containers e imagens

Conclusão da seção

Volumes

Introdução da seção

O que são volumes?

- Uma **forma prática de persistir dados** em aplicações e não depender de containers para isso;
- **Todo dado criado por um container é salvo nele**, quando o container é removido perdemos os dados;
- Então precisamos dos volumes para gerenciar os dados e também conseguir **fazer backups** de forma mais simples;

Tipos de volumes

- **Anônimos (anonymous):** Diretórios criados pela flag -v, porém com um nome aleatório;
- **Nomeados (named):** São volumes com nomes, podemos nos referir a estes facilmente e saber para que são utilizados no nosso ambiente;
- **Bind mounts:** Uma forma de salvar dados na nossa máquina, sem o gerenciamento do Docker, informamos um diretório para este fim;

O problema da persistência

- Se criarmos um container com alguma imagem, **todos os arquivos que geramos dentro dele serão do container**;
- Quando o container for removido, perderemos estes arquivos;
- Por isso precisamos os **volumes**;
- Vamos criar um exemplo prático!

Volumes anônimos

- Podemos criar um volume anônimo (**anonymous**) da seguinte maneira: **docker run -v /data**
- Onde **/data** será o diretório que contém o volume anônimo;
- E este container estará atrelado ao volume anônimo;
- Com o comando **docker volume ls**, podemos ver todos os volumes do nosso ambiente;

Volumes nomeados

- Podemos criar um volume nomeado (**named**) da seguinte maneira:

  docker run -v nomedovolume:/data
- Agora o volume tem um nome e pode ser facilmente referenciado;
- Em **docker volume ls** podemos verificar o container nomeado criado;
- Da mesma maneira que o anônimo, este volume tem como função armazenar arquivos;

Bind mounts

- **Bind mount** também é um volume, porém ele fica em um diretório que nós especificamos;
- Então não criamos um volume em sim, **apontamos um diretório**;
- O comando para criar um bind mount é: **docker run /dir/data:/data**
- Desta maneira o diretório **/dir/data** no nosso computador, será o volume deste container;

Atualização do projeto com bind mount

- **Bind mount** não serve apenas para volumes!
- Podemos utilizar esta técnica para **atualização em tempo real do projeto**;
- Sem ter que refazer o build a cada atualização do mesmo;
- Vamos ver na prática!

Criar um volume

- Podemos criar volumes manualmente também;
- Utilizamos o comando: **docker volume create <nome>**;
- Desta maneira temos um **named volume** criado, podemos atrelar a algum container na execução do mesmo;

Listando todos os volumes

- Com o comando: **docker volume ls** listamos todos todos os volumes;
- Desta maneira temos acesso aos **anonymous e os named volumes**;
- Interessante para saber quais volumes estão criados no nosso ambiente;

Checar um volume

- Podemos verificar os detalhes de um volume em específico com o comando: **docker volume inspect nome**;
- Desta forma temos acesso ao **local em que o volume guarda dados**, nome, escopo e muito mais;
- O docker salva os dados dos volumes em algum diretório do nosso computador, desta forma podemos saber qual é;

Remover um volume

- Podemos também remover um volume existente de forma fácil;
- Vamos utilizar o comando **docker volume rm <nome>**
- Observe que **os dados serão removidos todos também**, tome cuidado com este comando;

Removendo volumes não utilizados

- Podemos **remover todos os volumes que não estão sendo utilizados** com apenas um comando;
- O comando é: **docker volume prune**
- Semelhante ao prune que remove imagens e containers, visto anteriormente;

Volume apenas de leitura

- Podemos criar um volume que tem **apenas permissão de leitura**, isso é útil em algumas aplicações;
- Para realizar esta configuração devemos utilizar o comando: **docker run -v volume:/data:ro**
- Este **:ro** é a abreviação de read only;

Volumes

Conclusão da seção

Networks

Introdução da seção

O que são Networks no Docker?

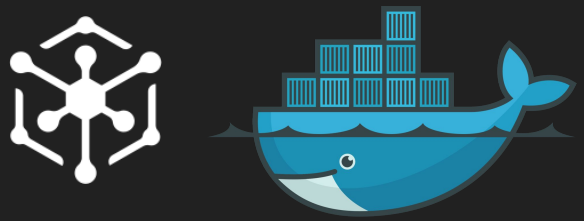
- **Uma forma de gerenciar a conexão do Docker** com outras plataformas ou até mesmo entre containers;
- As redes ou networks são **criadas separadas do containers**, como os volumes;
- Além disso existem alguns **drivers de rede**, que veremos em seguida;
- Uma rede deixa muito simples a comunicação entre containers;

Tipos de rede (drivers)

- **Bridge:** o mais comum e default do Docker, utilizado quando containers precisam se conectar (na maioria das vezes optamos por este driver);
- **host:** permite a conexão entre um container a máquina que está hosteando o Docker;
- **macvlan:** permite a conexão a um container por um MAC address;
- **none:** remove todas conexões de rede de um container;
- **plugins:** permite extensões de terceiros para criar outras redes;

Tipos de conexão

- Os containers costumam ter três principais tipos de comunicação:
- **Externa:** conexão com uma API de um servidor remoto;
- **Com o host:** comunicação com a máquina que está executando o Docker;
- **Entre containers:** comunicação que utiliza o driver bridge e permite a comunicação entre dois ou mais containers;



Listando redes

- Podemos verificar todas as redes do nosso ambiente com: **docker network ls**;
- **Algumas redes já estão criadas**, estas fazem parte da configuração inicial do docker;

Criando rede

- Para criar uma rede vamos utilizar o comando docker **network create <nome>**;
- Esta rede será do tipo **bridge**, que é o mais utilizado;
- Podemos criar diversas redes;

Removendo redes

- Podemos remover redes de forma simples também: **docker network rm <nome>**;
- Assim **a rede não estará mais disponível** para utilizarmos;
- Devemos tomar cuidado com containers já conectados;

Removendo redes em massa

- Podemos remover redes de forma simples também: **docker network prune**;
- Assim **todas as redes não utilizadas** no momento serão removidas;
- Receberemos uma mensagem de confirmação do Docker antes da ação ser executada;

Instalação do Postman

- Vamos criar uma **API** para testar a conexão entre containers;
- Para isso vamos utilizar o software **Postman**, que é o mais utilizado do mercado para desenvolvimento de APIs;
- Link: **https://www.postman.com/**

Conexão externa

- Os containers **podem se conectar livremente ao mundo externo**;
- Um caso seria: uma API de código aberto;
- Podemos acessá-la livremente e utilizar seus dados;
- Vamos testar!

Conexão com o host

- Podemos também **conectar um container com o host do Docker**;
- **Host** é a máquina que está executando o Docker;
- Como ip de host utilizamos: **host.docker.internal**
- No caso pode ser a nossa mesmo! =)

Conexão entre containers

- Podemos também estabelecer uma **conexão entre containers**;
- Duas imagens distintas rodando em **containers separados que precisam se conectar para inserir um dado no banco**, por exemplo;
- Vamos precisar de uma rede **bridge**, para fazer esta conexão;
- Agora nosso container de flask vai inserir dados em um MySQL que roda pelo Docker também;

Conectar container

- Podemos conectar um container a uma rede;
- Vamos utilizar o comando **docker network connect <rede> <container>**
- Após o comando o container estará dentro da rede!

Desconectar container

- Podemos desconectar um container a uma rede também;
- Vamos utilizar o comando **docker network disconnect <rede> <container>**
- Após o comando o container estará fora da rede!

Inspecionando redes

- Podemos analisar os detalhes de uma rede com o comando: **docker network inspect <none>**
- Vamos receber informações como: data de criação, driver, nome e muito mais!

Networks

Conclusão da seção

YAML

Introdução da seção

O que é YAML?

- Uma linguagem de serialização, seu nome é **YAML ain't Markup Language** (YAML não é uma linguagem de marcação);
- Usada geralmente para arquivos de configuração, inclusive do Docker, para configurar o **Docker Compose**;
- É de fácil leitura para nós humanos;
- A extensão dos arquivos é **yml** ou **yaml**;

Vamos criar nosso arquivo YAML

- O arquivo **.yaml** geralmente possui chaves e valores;
- Que é de onde vamos retirar as configurações do nosso sistema;
- Para definir uma chave apenas inserimos o nome dela, em seguida colocamos **dois pontos e depois o valor**;
- Vamos criar nosso primeiro arquivo YAML!

Espaçamento e indentação

- O **fim de uma linha** indica o fim de uma instrução, não há ponto e vírgula;
- A indentação deve conter **um ou mais espaços**, e não devemos utilizar tab;
- E cada uma define um novo bloco;
- O **espaço é obrigatório** após a declaração da chave;
- Vamos ver na prática!

Comentários

- Podemos escrever comentários em YAML também, utilizando o símbolo **#**;
- O processador de YAML **ignora comentários**;
- Eles são úteis para escrever como o arquivo funciona/foi configurado;

Dados numéricos

- Em YAML podemos escrever dados numéricos com:
- **Inteiros** = 12;
- **Floats** = 15.8;

Strings

- Em YAML podemos escrever textos de duas formas:
- **Sem aspas**: este é um texto válido
- **Com aspas**: “e este também”

Dados nulos

- Em YAML podemos definir um dado como nulo de duas formas:
- **~** ou **null**
- Os dois vão resultar em None, após a interpretação

Booleanos

- Podemos inserir booleanos em YAML da seguinte forma:
- **True e On** = verdadeiro;
- **False e Off** = falso;

Arrays

- Os arrays, tipos de dados para listas, possuem duas sintaxes:
- Primeira: **[1, 2, 3, 4, 5]**
- Segunda:

items:

- **1**
- **2**
- **3**

Dicionários

- Os dicionários, tipo de dados para objetos ou listas com chaves e valores, podem ser escritos assim:
- obj: {a: 1, b: 2, c: 3}
- E também com o nesting:

objeto:

    chave: 1

    chave: 2

YAML

Conclusão da seção

Docker Compose

Introdução da seção

O que é o Docker Compose?

- O **Docker Compose** é uma ferramenta para rodar múltiplos containers;
- Teremos apenas um arquivo de configuração, que orquestra totalmente esta situação;
- É uma forma de rodar **múltiplos builds e runs** com um comando;
- Em **projetos maiores** é essencial o uso do Compose;

Instalando docker compose no Linux

- **Usuários do Linux** ainda não possuem a ferramenta que utilizaremos nesta seção;
- Vamos seguir as instruções de: https://docs.docker.com/compose/install/
- O **docker compose** é essencial para atingirmos os nossos objetos no curso;

Criando nosso primeiro Compose

- Primeiramente vamos criar um arquivo chamado **docker-compose.yml** na raiz do projeto;
- Este arquivo vai **coordenar os containers e imagens**, e possui algumas chaves muito utilizadas;
- **version:** versão do Compose;
- **services:** Containers/serviços que vão rodar nessa aplicação;
- **volumes:** Possível adição de volumes;

Rodando o Compose

- Para rodar nossa estrutura em Compose vamos utilizar o comando: **docker compose up**;
- Isso fará com que as **instruções no arquivo sejam executadas**;
- Da mesma forma que realizamos os builds e também os runs;
- Podemos parar o Compose com **ctrl+c** no terminal;

Compose em background

- O Compose também pode ser executado em modo **detached**;
- Para isso vamos utilizar a **flag -d** no comando;
- E então os containers estarão **rodando em background;**
- Podemos ver sua execução com **docker ps**;

Parando o Compose

- Podemos parar o Compose que roda em background com: **docker compose down**;
- Desta maneira o serviço para e temos os containers adicionados no **docker ps -a**;

Variáveis de ambiente

- Podemos definir variáveis de ambiente para o Docker Compose;
- Para isso vamos definir um arquivo base em **env_file**;
- As variáveis podem ser chamadas pela sintaxe: **${VARIAVEL}**
- Esta técnica é útil quando o dado a ser inserido é **sensível/não pode ser compartilhado**, como uma senha;

Redes no Compose

- O Compose cria uma **rede básica Bridge** entre os containers da aplicação;
- Porém podemos isolar as redes com a chave **networks**;
- Desta maneira podemos conectar apenas os containers que optarmos;
- E podemos **definir drivers diferentes** também;

Vamos incluir o projeto no Compose

- Agora vamos inserir o nosso projeto da última seção no Compose;
- Para verificar na prática como fazer uma transferência de **Dockerfiles para Docker Compose!**

Build no Compose

- Podemos gerar o **build durante o Compose** também;
- Isso vai **eliminar o processo de gerar o build da imagem** a cada atualização;
- Vamos ver na prática!

Bind mount no Compose

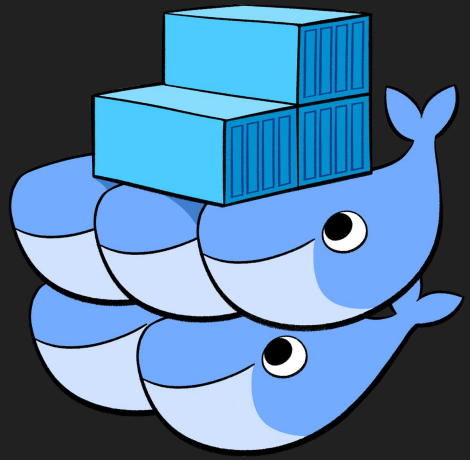
- O volume de **Bind Mount garante atualização em tempo real dos arquivos do container**;
- Podemos configurar nosso projeto de Compose para utilizar esta funcionalidade também;
- Vamos ver na prática!

Verificando o que tem no Compose

- Podemos fazer a verificação do compose com: **docker-compose ps**
- Receberemos um **resumo dos serviços que sobem** ao rodar o compose;
- Desta maneira podemos avaliar rapidamente o projeto;

Docker Compose

Conclusão da seção

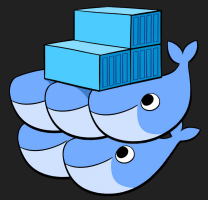


Docker Swarm

Introdução da seção

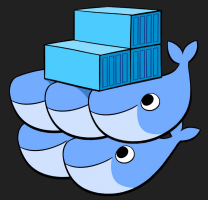
O que é orquestração de containers?

- Orquestração é o ato de conseguir **gerenciar e escalar** os containers da nossa aplicação;
- Temos **um serviço que rege sobre outros serviços**, verificando se os mesmos estão funcionando como deveriam;
- Desta forma conseguimos garantir uma aplicação saudável e também que esteja sempre disponível;
- Alguns serviços: **Docker Swarm, Kubernetes e Apache Mesos**;



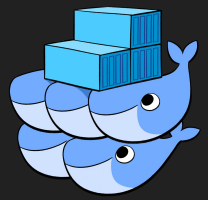
O que é Docker Swarm?

- Uma ferramenta do Docker para **orquestrar containers**;
- Podendo **escalar horizontalmente** nossos projetos de maneira simples;
- O famoso **cluster**!
- A **facilidade do Swarm** para outros orquestradores é que todos os comandos são muito semelhantes ao do Docker;
- Toda instalação do Docker já vem com Swarm, **porém desabilitado**;



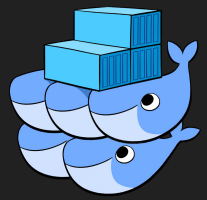
Conceitos fundamentais

- **Nodes:** é uma instância (máquina) que participa do Swarm;
- **Manager Node:** Node que gerencia os demais Nodes;
- **Worker Node:** Nodes que trabalham em função do Manager;
- **Service:** Um conjunto de Tasks que o Manager Node manda o Work Node executar;
- **Task:** comandos que são executados nos Nodes;



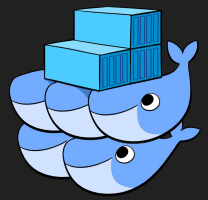
Maneira de executar o Swarm

- Para exemplificar corretamente o Swarm vamos precisar de Nodes, ou seja, **mais máquinas**;
- Então temos duas soluções:
- **AWS**, criar a conta e rodar alguns servidores (precisa de cartão de crédito, mas é gratuito);
- **Docker Labs**, gratuito também, roda no navegador, porém expira a cada 4 horas;



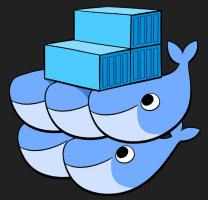
Iniciando o Swarm

- Podemos iniciar o Swarm com o comando: **docker swarm init**;
- Em alguns casos precisamos declarar o IP do servidor com a flag: **--advertise-addr**
- Isso fará com que a instância/máquina vire um **Node**;
- E também transforma o Node em um **Manager**;



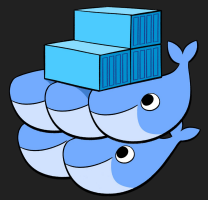
Listando Nodes ativos

- Podemos verificar quais Nodes estão ativos com: **docker node ls**
- Desta forma os serviços serão exibidos no terminal;
- Podemos assim **monitorar o que o Swarm está orquestrando**;
- Este comando será de grande utilidade a medida que formos adicionando serviços no Swarm;



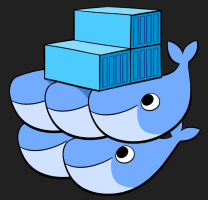
Adicionando novos Nodes

- Podemos adicionar um novo serviço com o comando: **docker swarm join --token <TOKEN> <IP>:<PORTA>**
- Desta forma duas máquinas estarão conectadas;
- Esta nova máquina entra na hierarquia como **Worker**;
- Todas as ações (**Tasks**) utilizadas na Manager, serão replicadas em Nodes que foram adicionados com join;



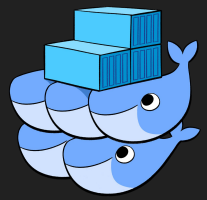
Subindo um novo serviço

- Podemos iniciar um serviço com o comando: **docker service create --name <nome> <imagem>**
- Desta forma teremos um container novo sendo adicionado ao nosso Manager;
- E este serviço estará sendo gerenciado pelo Swarm;
- Podemos testar com o nginx, **liberando a porta 80** o container já pode ser acessado;



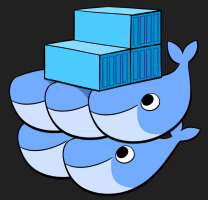
Listando serviços

- Podemos listar os serviços que estão rodando com: **docker service ls**
- Desta maneira todos os serviços que iniciamos serão exibidos;
- Algumas informações importantes sobre eles estão na tabela: **nome, replicas, imagem, porta**;



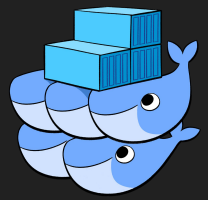
Removendo serviços

- Podemos remover um serviço com: **docker service rm <nome>**
- Desta maneira o serviço para de rodar;
- Isso pode significar: **parar um container que está rodando** e outras consequências devido a parada do mesmo;
- Checamos a remoção com: docker service ls



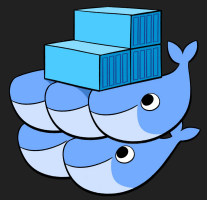
Aumentando o número de réplicas

- Podemos criar um serviço com um número maior de réplicas: **docker service create --name <NOME> --replicas <NUMERO> <IMAGEM>**
- Desta maneira uma task será emitida, replicando este serviço nos Workers;
- Agora iniciamos de fato a **orquestração**;
- Podemos checar o status com: docker service ls



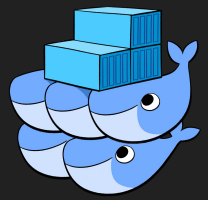
Verificando a orquestração

- Vamos remover um container de um **Node Worker;**
- Isso fará com que o Swarm reinicie este container novamente;
- Pois o serviço ainda está rodando no **Manager**, e isto é uma de suas atribuições: **garantir que os serviços estejam sempre disponíveis**;
- Obs: precisamos utilizar o force (-f);



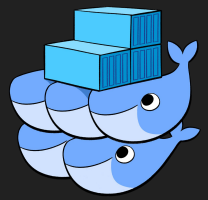
Checando token do Swarm

- As vezes vamos precisar checar o token do Swarm, **para dar join em alguma outra instância futuramente**;
- Então temos o comando: **docker swarm join-token manager**
- Desta forma recebemos o token pelo terminal;



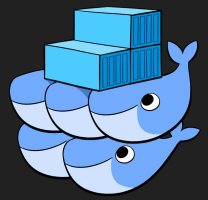
Checando o Swarm

- Podemos verificar detalhes do Swarm que o Docker está utilizando;
- Utilizamos o comando: **docker info**;
- Desta forma recebemos informações como: I**D do Node, número de nodes, número de managers e muito mais!**



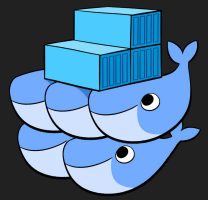
Removendo instância do Swarm

- Podemos parar de executar o Swarm em uma determinada instância também;
- Vamos utilizar o comando: **docker swarm leave**
- A partir deste momento, **a instância não é contada mais como um Node para o Swarm**;



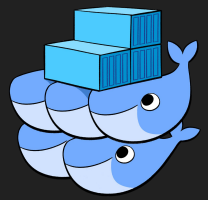
Removendo um Node

- Podemos também remover um Node do nosso ecossistema do Swarm;
- Vamos utilizar o comando: **docker node rm <ID>**
- **Desta forma a instância não será considerada mais um Node**, saindo do Swarm;
- O container continuará rodando na instância;
- Precisamos utilizar o -f



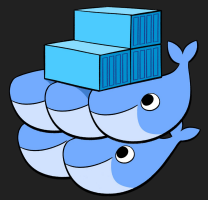
Inspecionando serviços

- Podemos ver em mais detalhes o que um serviço possui;
- O comando é: **docker service inspect <ID>**
- Vamos receber informações como: **nome, data de criação, portas e etc**;



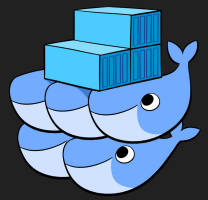
Verificar containers ativados pelo service

- Podemos ver quais containers um serviço já rodou:
- O comando é: **docker service ps <ID>**
- Receberemos uma lista de containers que estão rodando e também dos que já receberam baixa;
- Este comando é **semelhante ao docker ps -a**;



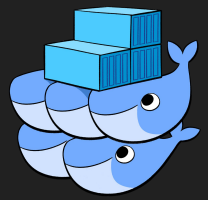
Rodando Compose com Swarm

- Para rodar Compose com Swarm vamos utilizar os comandos de Stack;
- O comando é: **docker stack deploy -c <ARQUIVO.YAML> <NOME>**
- Teremos então o **arquivo compose** sendo executado;
- Porém agora estamos em modo swarm e podemos utilizar os Nodes como réplicas;



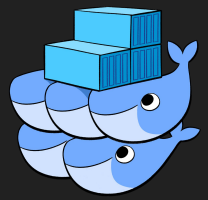
Aumentando réplicas do Stack

- Podemos criar novas réplicas nos **Worker Nodes**;
- Vamos utilizar o comando: **docker service scale <NOME>=<REPLICAS>**
- Desta forma as outras máquinas receberão as Tasks a serem executadas;



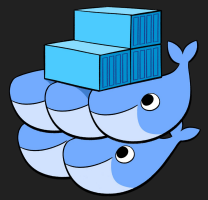
Fazer serviço não receber mais Tasks

- Podemos fazer com que um serviço **não receba mais ‘ordens’ do Manager**;
- Para isso vamos utilizar o comando: **docker node update --availability drain <ID>**
- O status de drain, é o que não recebe tasks;
- Podemos voltar para **active**, e ele volta ao normal;



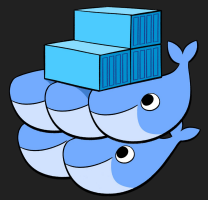
Atualizar parâmetro

- Podemos atualizar as configurações dos nossos nodes;
- Vamos utilizar o comando: **docker service update --image <IMAGEM> <SERVICO>**
- Desta forma apenas os nodes que estão com o status **active** receberão atualizações;



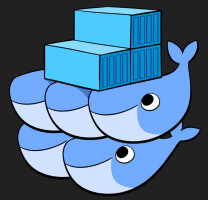
Criando rede para Swarm

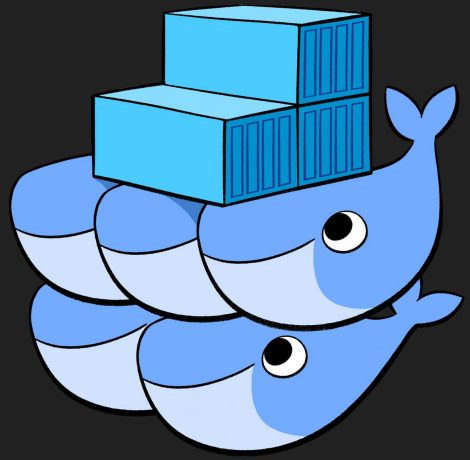
- A conexão entre instâncias usa um driver diferente, o **overlay**;
- Podemos criar primeiramente a rede com **docker network create**;
- E depois ao criar um service adicionar a flag **--network <REDE>** para inserir as instâncias na nossa nova rede;



Conectar serviço a uma rede

- Podemos também conectar serviços que já estão em execução a uma rede;
- Vamos utilizar o comando de update: **docker service update --network <REDE> <NOME>**
- Depois checamos o resultado com **inspect**;





Docker Swarm

Conclusão da seção

Kubernetes

Introdução da seção

O que é Kubernetes?

- Uma ferramenta de **orquestração de containers**;
- Permite a criação de **múltiplos containers em diferentes máquinas (nodes)**;
- Escalando projetos, formando um **cluster**;
- Gerencia serviços, garantindo que as aplicações sejam executadas **sempre da mesma forma**;
- Criada pelo **Google**;

Conceitos fundamentais

- **Control Plane:** Onde é gerenciado o controle dos processos dos Nodes;
- **Nodes:** Máquinas que são gerenciadas pelo Control Plane;
- **Deployment:** A execução de uma imagem/projeto em um Pod;
- **Pod:** um ou mais containers que estão em um Node;
- **Services:** Serviços que expõe os Pods ao mundo externo;
- **kubectl:** Cliente de linha de comando para o Kubernetes;

Dependências necessárias

- O Kubernetes pode ser executado de uma maneira simples em nossa máquina;
- Vamos precisar do client, **kubectl**, que é a maneira de executar o Kubernetes;
- E também o **Minikube**, uma espécie de simulador de Kubernetes, para não precisarmos de vários computadores/servidores;

Kubernetes no Windows

- Primeiramente vamos instalar o gerenciador de pacotes **Chocolatey**;
- Depois seguiremos a documentação de instalação do **client de Kubernetes**;
- Devemos também instalar o **Virtualbox** (não é necessário se você tem o Hyper-V ou o Docker instalado);
- E por fim o **Minikube**;
- Na próxima aula vamos inicializar o Minikube!

Kubernetes no Linux

- No Linux vamos instalar primeiramente o **client do Kubernetes**, seguindo a documentação;
- E depois também seguiremos a documentação do **Minikube**;
- Um dos requisitos do Minikube é ter um **gerenciador de VMs/containers** como: Docker, Virtual box, Hiper-V;
- Na próxima aula vamos inicializar o Minikube!

Iniciando o Minikube

- Para inicializar o Minikube vamos utilizar o comando: **minikube start --driver=<DRIVER>**
- Onde o driver vai depender de como foi sua instalação das dependências, e por qualquer um deles atingiremos os mesmos resultados!
- Você pode tentar: **virtualbox, hyperv e docker**
- Podemos testar o Minikube com: **minikube status**

Parando o Minikube

- **Obs:** sempre que o computador for reiniciado, deveremos iniciar o Minikube;
- E podemos pará-lo também com o comando: **minikube stop**
- Para iniciar novamente, digite: **minikube start --driver=<DRIVER>**

Acessando a dashboard do Kubernetes

- O Minikube nos disponibiliza uma **dashboard**;
- Nela podemos ver todo o detalhamento de nosso projeto: **serviços, pods e etc**;
- Vamos acessar com o comando: **minikube dashboard**
- Ou para apenas obter a URL: **minibuke dashboard --url**

Deployment teoria

- O **Deployment** é uma parte fundamental do Kubernetes;
- Com ele criamos nosso serviço que vai rodar nos **Pods**;
- **Definimos uma imagem e um nome**, para posteriormente ser replicado entre os servidores;
- A partir da criação do deployment teremos containers rodando;
- Vamos precisar de uma **imagem no Hub do Docker**, para gerar um Deployment;

Criar projeto

- Primeiramente vamos criar um pequeno projeto, novamente em **Flask**;
- Buildar a **imagem** do mesmo;
- Enviar a imagem para o **Docker Hub**;
- E testar rodar em um **container**;
- Este projeto será utilizado no **Kubernetes**!

Criando nosso Deployment

- Após este mini setup é hora de rodar nosso projeto no **Kubernetes**;
- Para isso vamos precisar de um **Deployment**, que é onde rodamos os containers das aplicações nos **Pods**;
- O comando é: **kubectl create deployment <NOME> --image=<IMAGEM>**
- Desta maneira o projeto de Flask estará sendo orquestrado pelo Kubernetes;

Checando Deployments

- Podemos checar se tudo foi criado corretamente, tanto o **Deployment** quanto a recepção do projeto pelo **Pod**;
- Para verificar o Deployment vamos utilizar: **kubectl get deployments**
- E para receber mais detalhes deles: **kubectl describe deployments**
- Desta forma conseguimos **saber se o projeto está de fato rodando** e também **o que está rodando nele**;

Checando Pods

- Os **Pods** são componentes muito importantes também, onde os containers realmente são executados;
- Para verificar os Pods utilizamos: **kubectl get pods**
- E para saber mais detalhes deles: **kubectl describe pods**
- Recebemos **o status dos Pods** que estão ligados e também **informações importantes sobre eles**;

Configurações do Kubernetes

- Podemos também verificar **como o Kubernetes está configurado**;
- O comando é: **kubectl config view**
- No nosso caso: vamos receber informações importantes baseadas no **Minikube**, que é por onde o Kubernetes está sendo executado;

Services teoria

- As aplicações do Kubernetes **não tem conexão com o mundo externo**;
- Por isso precisamos criar um **Service**, que é o que possibilita expor os Pods;
- Isso acontece pois os **Pods são criados para serem destruídos** e perderem tudo, ou seja, os dados gerados neles também são apagados;
- Então o **Service é uma entidade separada dos Pods**, que expõe eles a uma rede;

Criando nosso Service

- Para criar um serviço e expor nossos Pods devemos utilizar o comando: **kubectl expose deployment <NOME> --type=<TIPO> --port=<PORT>**
- Colocaremos o nome do **Deployment** já criado;
- O **tipo de Service**, há vários para utilizarmos, porém o **LoadBalancer** é o mais comum, onde todos os Pods são expostos;
- E uma **porta** para o serviço ser consumido;

Gerando Ip de acesso

- Podemos acessar o nosso serviço com o comando: **minikube service <NOME>**
- Desta forma o **IP aparece no nosso terminal**;
- E também **uma aba no navegador é aberta** com o projeto;
- E pronto! Temos um projeto rodando pelo **Kubernetes**!

Verificando os nosso serviços

- Podemos também obter detalhes dos **Services já criados**;
- O comando para verificar todos é: **kubectl get services**
- E podemos obter informações de um serviço em específico com: **kubectl describe services/<NOME>**

Replicando nossa aplicação

- Vamos aprender agora a como utilizar outros Pods, **replicando assim a nossa aplicação**;
- O comando é: **kubectl scale deployment/<NOME> --replicas=<NUMERO>**
- Podemos agora verificar no **Dashboard** o aumento de Pods;
- E também com o comando de: **kubectl get pods**

Checar número de réplicas

- Além do get pods e da Dashboard, temos mais um comando para **checar réplicas**;
- Que é o: **kubectl get rs**
- Desta maneira temos os **status das réplicas** dos projetos;

Diminuindo a escala

- Podemos facilmente também **reduzir o número de Pods**;
- Esta técnica é chamada de **scale down**;
- O comando é o mesmo, porém colocamos menos réplicas e o Kubernetes faz o resto;
- Comando: **kubectl scale deployment/<NOME> --replicas=<NUMERO_MENOR>**

Resgatar o IP do serviço

- Podemos sempre relembrar o IP/URL do nosso serviço;
- O comando é: **minikube service --url <NOME>**
- Desta maneira **a URL é exibida no terminal**;

Atualização de imagem

- Para atualizar a imagem vamos precisar do **nome do container**, isso é dado na Dashboard dentro do Pod;
- E também a nova imagem deve ser uma outra versão da atual, **precisamos subir uma nova tag no Hub**;
- Depois utilizamos o comando: **kubectl set image deployment/<NOME> <NOME_CONTAINER>=<NOVA_IMAGEM>**

Desfazer alteração

- Para desfazer uma alteração utilizamos uma ação conhecida como rollback;
- O comando para verificar uma alteração é: **kubectl rollout status deployment/<NOME>**
- Com ele e com o **kubectl get pods**, podemos identificar problemas;
- Para voltar a alteração utilizamos: **kubectl rollout undo deployment/<NOME>**

Desfazer alteração

- Para desfazer uma alteração utilizamos uma ação conhecida como rollback;
- O comando para verificar uma alteração é: **kubectl rollout status deployment/<NOME>**
- Com ele e com o **kubectl get pods**, podemos identificar problemas;
- Para voltar a alteração utilizamos: **kubectl rollout undo deployment/<NOME>**

Deletar um Service

- Para deletar um serviço do Kubernetes vamos utilizar o comando: **kubectl delete service <NOME>**
- Desta maneira nossos Pods **não terão mais a conexão externa**;
- Ou seja, não poderemos mais acessar eles;

Deletar um Deployment

- Para deletar um Deployment do Kubernetes vamos utilizar o comando: **kubectl delete deploymnet <NOME>**
- Desta maneira **o container não estará mais rodando**, pois paramos os Pods;
- Assim precisaremos criar um deployment novamente com a mesma ou outra imagem, para acessar algum projeto;

Modo declarativo

- Até agora utilizamos o **modo imperativo**, que é quando iniciamos a aplicação com comandos;
- O **modo declarativo** é guiado por um arquivo, semelhante ao **Docker Compose**;
- Desta maneira tornamos nossas configurações mais simples e **centralizamos tudo em um comando**;
- Também escrevemos em **YAML** o arquivo de Kubernetes;

Chaves mais utilizadas

- **apiVersion**: versão utilizada da ferramenta;
- **kind**: tipo do arquivo (Deployment, Service);
- **metadata**: descrever algum objeto, inserindo chaves como name;
- **replicas**: número de réplicas de Nodes/Pods;
- **containers**: definir as especificações de containers como: nome e imagem;

Criando nosso arquivo

- Agora vamos transformar nosso projeto em **declarativo**;
- Para isso vamos criar um arquivo para realizar o **Deployment**;
- Desta maneira vamos aprender a criar os arquivos declarativos e utilizar as **chaves e valores**;
- Mãos à obra!

Executando arquivo de Deployment

- Vamos então executar nosso arquivo de **Deployment**!
- O comando é: **kubectl apply -f <ARQUIVO>**
- Desta maneira o Deployment será criado conforme configurado no arquivo .yaml;

Parando o Deployment

- Para parar de executar este deployment baseado em arquivo, o **declarativo**, utilizamos também o delete;
- O comando é: **kubectl delete -f <ARQUIVO>**
- Desta maneira teremos os Pods sendo excluídos e o serviço finalizado;

Criando o serviço

- Agora vamos criar o serviço em **declarativo**;
- Para isso vamos criar um arquivo para realizar o **Service (kind)**;
- O arquivo será semelhante ao de Deployment, porém tem uma responsabilidade diferente;

Executando o serviço

- Vamos executar da mesma maneira: **kubectl apply -f <ARQUIVO>**
- E o serviço vai estar disponível;
- Obs: precisamos gerar o IP de acesso com **minikube service <NOME>**

Parando o Serviço

- Para parar de executar um serviço baseado em arquivo, o **declarativo**, utilizamos também o delete;
- O comando é: **kubectl delete -f <ARQUIVO>**
- Desta maneira o serviço não estará mais disponível, então perdemos o acesso ao projeto;

Atualizando o projeto no declarativo

- Primeiramente vamos **criar uma nova versão da imagem;**
- E fazer o **push para o Hub**;
- Depois é só alterar no arquivo de Deployment a **tag**;
- E reaplicar o comando de **apply**, simples assim! =)

Unindo arquivos do projeto

- Vamos precisar unir o deployment e o service em um arquivo;
- A separação de objetos para o YAML é com: **---**
- Desta forma cada um deles será executado;
- Uma boa prática é colocar o **service antes do deployment**!

Kubernetes

Introdução da seção