

DAY-1

O quê preciso saber antes de começar?

Durante o Day-1 nós vamos entender o que é um container, vamos falar sobre a importância do container runtime e do container engine. Durante o Day-1 vamos entender o que é o Kubernetes e sua arquitetura, vamos falar sobre o control plane, workers, apiserver, scheduler, controller e muito mais! Será aqui que iremos criar o nosso primeiro cluster Kubernetes e realizar o deploy de um pod do Nginx. O Day-1 é para que eu possa me sentir mais confortável com o Kubernetes e seus conceitos iniciais.

Início da aula do Day-1

Qual distro GNU/Linux devo usar?

Devido ao fato de algumas ferramentas importantes, como o `systemd` e `journald`, terem se tornado padrão na maioria das principais distribuições disponíveis hoje, você não deve encontrar problemas para seguir o treinamento, caso você opte por uma delas, como Ubuntu, Debian, CentOS e afins.

Alguns sites que devemos visitar

Abaixo temos os sites oficiais do projeto do Kubernetes:

- <https://kubernetes.io>
- <https://github.com/kubernetes/kubernetes/>
- <https://github.com/kubernetes/kubernetes/issues>

Abaixo temos as páginas oficiais das certificações do Kubernetes (CKA, CKAD e CKS):

- <https://www.cncf.io/certification/cka/>
- <https://www.cncf.io/certification/ckad/>
- <https://www.cncf.io/certification/cks/>

O Container Engine

Antes de começar a falar um pouco mais sobre o Kubernetes, nós primeiro precisamos entender alguns componentes que são importantes no ecossistema do Kubernetes, um desses componentes é o Container Engine.

O *Container Engine* é o responsável por gerenciar as imagens e volumes, é ele o responsável por garantir que os recursos que os containers estão utilizando está devidamente isolados, a vida do container, storage, rede, etc.

Hoje temos diversas opções para se utilizar como *Container Engine*, que até pouco tempo atrás tínhamos somente o Docker para esse papel.

Opções como o Docker, o CRI-O e o Podman são bem conhecidas e preparadas para o ambiente produtivo. O Docker, como todos sabem, é o Container Engine mais popular e ele utiliza como Container Runtime o containerd.

Container Runtime? O que é isso?

Calma que vou te explicar já já, mas antes temos que falar sobre a OCI. :)

OCI - Open Container Initiative

A OCI é uma organização sem fins lucrativos que tem como objetivo padronizar a criação de containers, para que possam ser executados em qualquer ambiente. A OCI foi fundada em 2015 pela Docker, CoreOS, Google, IBM, Microsoft, Red Hat e VMware e hoje faz parte da Linux Foundation.

O principal projeto criado pela OCI é o *runc*, que é o principal container runtime de baixo nível, e utilizado por diferentes *Container Engines, como o Docker. O *runc* é um projeto open source, escrito em Go e seu código está disponível no GitHub.

Agora sim já podemos falar sobre o que é o Container Runtime.

O Container Runtime

Para que seja possível executar os containers nós é necessário ter um *Container Runtime* instalado em cada um dos nós.

O *Container Runtime* é o responsável por executar os containers nos nós. Quando você está utilizando Docker ou Podman para executar containers em sua máquina, por exemplo, você está fazendo uso de algum *Container Runtime*, ou melhor, o seu Container Engine está fazendo uso de algum *Container Runtime*.

Temos três tipos de *Container Runtime*:

- Low-level: são os *Container Runtime* que são executados diretamente pelo Kernel, como o runc, o crun e o runsc.
- High-level: são os *Container Runtime* que são executados por um *Container Engine*, como o containerd, o CRI-O e o Podman.
- Sandbox: são os *Container Runtime* que são executados por um *Container Engine* e que são responsáveis por executar containers de forma segura em unikernels ou utilizando algum proxy para fazer a comunicação com o Kernel. O gVisor é um exemplo de *Container Runtime* do tipo Sandbox.
- Virtualized: são os *Container Runtime* que são executados por um *Container Engine* e que são responsáveis por executar containers de forma segura em máquinas virtuais. A performance aqui é um pouco menor do que quando temos um sendo executado nativamente. O Kata Containers é um exemplo de *Container Runtime* do tipo Virtualized.

O que é o Kubernetes?

Versão resumida:

O projeto Kubernetes foi desenvolvido pela Google, em meados de 2014, para atuar como um orquestrador de contêineres para a empresa. O Kubernetes (k8s), cujo termo em Grego significa "timoneiro", é um projeto *open source* que conta com *design* e desenvolvimento baseados no projeto Borg, que também é da Google 1. Alguns outros produtos disponíveis no mercado, tais como o Apache Mesos e o Cloud Foundry, também surgiram a partir do projeto Borg.

Como Kubernetes é uma palavra difícil de se pronunciar - e de se escrever - a comunidade simplesmente o apelidou de k8s, seguindo o padrão i18n (a letra "k" seguida por oito letras e o "s" no final), pronunciando-se simplesmente "kates".

Versão longa:

Praticamente todo software desenvolvido na Google é executado em contêiner 2. A Google já gerencia contêineres em larga escala há mais de uma década, quando não

se falava tanto sobre isso. Para atender a demanda interna, alguns desenvolvedores do Google construíram três sistemas diferentes de gerenciamento de contêineres: Borg, Omega e Kubernetes. Cada sistema teve o desenvolvimento bastante influenciado pelo antecessor, embora fosse desenvolvido por diferentes razões.

O primeiro sistema de gerenciamento de contêineres desenvolvido no Google foi o Borg, construído para gerenciar serviços de longa duração e jobs em lote, que anteriormente eram tratados por dois sistemas: Babysitter e Global Work Queue. O último influenciou fortemente a arquitetura do Borg, mas estava focado em execução de jobs em lote. O Borg continua sendo o principal sistema de gerenciamento de contêineres dentro do Google por causa de sua escala, variedade de recursos e robustez extrema.

O segundo sistema foi o Omega, descendente do Borg. Ele foi impulsionado pelo desejo de melhorar a engenharia de software do ecossistema Borg. Esse sistema aplicou muitos dos padrões que tiveram sucesso no Borg, mas foi construído do zero para ter a arquitetura mais consistente. Muitas das inovações do Omega foram posteriormente incorporadas ao Borg.

O terceiro sistema foi o Kubernetes. Concebido e desenvolvido em um mundo onde desenvolvedores externos estavam se interessando em contêineres e o Google desenvolveu um negócio em amplo crescimento atualmente, que é a venda de infraestrutura de nuvem pública.

O Kubernetes é de código aberto - em contraste com o Borg e o Omega que foram desenvolvidos como sistemas puramente internos do Google. O Kubernetes foi desenvolvido com um foco mais forte na experiência de desenvolvedores que escrevem aplicativos que são executados em um cluster: seu principal objetivo é facilitar a implantação e o gerenciamento de sistemas distribuídos, enquanto se beneficia do melhor uso de recursos de memória e processamento que os contêineres possibilitam.

Estas informações foram extraídas e adaptadas deste artigo, que descreve as lições aprendidas com o desenvolvimento e operação desses sistemas.

Arquitetura do k8s

Assim como os demais orquestradores disponíveis, o k8s também segue um modelo *control plane/workers*, constituindo assim um *cluster*, onde para seu funcionamento é recomendado no mínimo três nós: o nó *control-plane*, responsável (por padrão) pelo gerenciamento do *cluster*, e os demais como *workers*, executores das aplicações que queremos executar sobre esse *cluster*.

É possível criar um cluster Kubernetes rodando em apenas um nó, porém é recomendado somente para fins de estudos e nunca executado em ambiente produtivo.

Caso você queira utilizar o Kubernetes em sua máquina local, em seu desktop, existem diversas soluções que irão criar um cluster Kubernetes, utilizando máquinas virtuais ou o Docker, por exemplo.

Com isso você poderá ter um cluster Kubernetes com diversos nós, porém todos eles rodando em sua máquina local, em seu desktop.

Alguns exemplos são:

- Kind: Uma ferramenta para execução de contêineres Docker que simulam o funcionamento de um cluster Kubernetes. É utilizado para fins didáticos, de desenvolvimento e testes. O Kind não deve ser utilizado para produção;
- Minikube: ferramenta para implementar um *cluster* Kubernetes localmente com apenas um nó. Muito utilizado para fins didáticos, de desenvolvimento e testes. O Minikube não deve ser utilizado para produção;
- MicroK8S: Desenvolvido pela Canonical, mesma empresa que desenvolve o Ubuntu. Pode ser utilizado em diversas distribuições e pode ser utilizado em ambientes de produção, em especial para *Edge Computing* e IoT (*Internet of things*);
- k3s: Desenvolvido pela Rancher Labs, é um concorrente direto do MicroK8s, podendo ser executado inclusive em Raspberry Pi.
- k0s: Desenvolvido pela Mirantis, mesma empresa que adquiriu a parte enterprise do Docker. É uma distribuição do Kubernetes com todos os recursos necessários para funcionar em um único binário, que proporciona uma simplicidade na instalação e manutenção do cluster. A pronúncia é correta é kay-zero-ess e tem por objetivo reduzir o esforço técnico e desgaste na instalação de um cluster Kubernetes, por isso o seu nome faz alusão a *Zero Friction*. O k0s pode ser utilizado em ambientes de produção/
- API Server: É um dos principais componentes do k8s. Este componente fornece uma API que utiliza JSON sobre HTTP para comunicação, onde para isto é utilizado principalmente o utilitário `kubectl`, por parte dos administradores, para a comunicação com os demais nós, como mostrado no gráfico. Estas comunicações entre componentes são estabelecidas através de requisições REST;
- etcd: O etcd é um *datastore* chave-valor distribuído que o k8s utiliza para armazenar as especificações, status e configurações do *cluster*. Todos os dados armazenados dentro do etcd são manipulados apenas através da API. Por questões de segurança, o etcd é por padrão executado apenas em nós

classificados como *control plane* no *cluster* k8s, mas também podem ser executados em *clusters* externos, específicos para o etcd, por exemplo;

- Scheduler: O *scheduler* é responsável por selecionar o nó que irá hospedar um determinado *pod* (a menor unidade de um *cluster* k8s - não se preocupe sobre isso por enquanto, nós falaremos mais sobre isso mais tarde) para ser executado. Esta seleção é feita baseando-se na quantidade de recursos disponíveis em cada nó, como também no estado de cada um dos nós do *cluster*, garantindo assim que os recursos sejam bem distribuídos. Além disso, a seleção dos nós, na qual um ou mais pods serão executados, também pode levar em consideração políticas definidas pelo usuário, tais como afinidade, localização dos dados a serem lidos pelas aplicações, etc;
- Controller Manager: É o *controller manager* quem garante que o *cluster* esteja no último estado definido no etcd. Por exemplo: se no etcd um *deploy* está configurado para possuir dez réplicas de um *pod*, é o *controller manager* quem irá verificar se o estado atual do *cluster* corresponde a este estado e, em caso negativo, procurará conciliar ambos;
- Kubelet: O *kubelet* pode ser visto como o agente do k8s que é executado nos nós workers. Em cada nó worker deverá existir um agente Kubelet em execução. O Kubelet é responsável por de fato gerenciar os *pods*, que foram direcionados pelo *controller* do *cluster*, dentro dos nós, de forma que para isto o Kubelet pode iniciar, parar e manter os contêineres e os pods em funcionamento de acordo com o instruído pelo controlador do cluster;
- Kube-proxy: Age como um *proxy* e um *load balancer*. Este componente é responsável por efetuar roteamento de requisições para os *pods* corretos, como também por cuidar da parte de rede do nó;

Portas que devemos nos preocupar

CONTROL PLANE

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443*	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd

TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

- Toda porta marcada por * é customizável, você precisa se certificar que a porta alterada também esteja aberta. WORKERS

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort	Services All

Conceitos-chave do k8s

É importante saber que a forma como o k8s gerencia os contêineres é ligeiramente diferente de outros orquestradores, como o Docker Swarm, sobretudo devido ao fato de que ele não trata os contêineres diretamente, mas sim através de *pods*. Vamos conhecer alguns dos principais conceitos que envolvem o k8s a seguir:

- Pod: É o menor objeto do k8s. Como dito anteriormente, o k8s não trabalha com os contêineres diretamente, mas organiza-os dentro de *pods*, que são abstrações que dividem os mesmos recursos, como endereços, volumes, ciclos de CPU e memória. Um pod pode possuir vários contêineres;
- Deployment: É um dos principais *controllers* utilizados. O *Deployment*, em conjunto com o *ReplicaSet*, garante que determinado número de réplicas de um pod esteja em execução nos nós workers do cluster. Além disso, o Deployment também é responsável por gerenciar o ciclo de vida das aplicações, onde características associadas a aplicação, tais como imagem, porta, volumes e

variáveis de ambiente, podem ser especificados em arquivos do tipo *yaml* ou *json* para posteriormente serem passados como parâmetro para o `kubectl` executar o deployment. Esta ação pode ser executada tanto para criação quanto para atualização e remoção do deployment;

- **ReplicaSets:** É um objeto responsável por garantir a quantidade de pods em execução no nó;
- **Services:** É uma forma de você expor a comunicação através de um *ClusterIP*, *NodePort* ou *LoadBalancer* para distribuir as requisições entre os diversos Pods daquele Deployment. Funciona como um balanceador de carga.

Instalando e customizando o Kubectl

Instalação do Kubectl no GNU/Linux

Vamos instalar o `kubectl` com os seguintes comandos.

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl
```

```
chmod +x ./kubectl
```

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

```
kubectl version --client
```

Instalação do Kubectl no MacOS

O `kubectl` pode ser instalado no MacOS utilizando tanto o Homebrew, quanto o método tradicional. Com o Homebrew já instalado, o `kubectl` pode ser instalado da seguinte forma.

```
sudo brew install kubectl
```

```
kubectl version --client
```

Ou:

```
sudo brew install kubectl-cli
```



```
kubectl version --client
```

Já com o método tradicional, a instalação pode ser realizada com os seguintes comandos.

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl"
```

```
chmod +x ./kubectl
```

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

```
kubectl version --client
```

Instalação do Kubectl no Windows

A instalação do `kubectl` pode ser realizada efetuando o download neste link.

Outras informações sobre como instalar o `kubectl` no Windows podem ser encontradas nesta página.

Customizando o kubectl

Auto-complete

Execute o seguinte comando para configurar o alias e autocomplete para o `kubectl`.

No Bash:

```
source <(kubectl completion bash) # configura o autocomplete na sua sessão atual (antes, certifique-se de ter instalado o pacote bash-completion).
```

```
echo "source <(kubectl completion bash)" >> ~/.bashrc # add autocomplete permanentemente ao seu shell.
```

No ZSH:

```
source <(kubectl completion zsh)
```

```
echo "[[ \$commands[kubectl] ]] && source <(kubectl completion zsh)"
```

Criando um alias para o kubectl

Crie o alias `k` para `kubectl`:

```
alias k=kubectl
```

```
complete -F __start_kubectl k
```

Criando um cluster Kubernetes

Criando o cluster em sua máquina local

Vamos mostrar algumas opções, caso você queira começar a brincar com o Kubernetes utilizando somente a sua máquina local, o seu desktop.

Lembre-se, você não é obrigado a testar/utilizar todas as opções abaixo, mas seria muito bom caso você testasse. :D

Minikube

Requisitos básicos

É importante frisar que o Minikube deve ser instalado localmente, e não em um *cloud provider*. Por isso, as especificações de *hardware* a seguir são referentes à máquina local.

- Processamento: 1 core;
- Memória: 2 GB;
- HD: 20 GB.

Instalação do Minikube no GNU/Linux

Antes de mais nada, verifique se a sua máquina suporta virtualização. No GNU/Linux, isto pode ser realizado com o seguinte comando:

```
grep -E --color 'vmx|svm' /proc/cpuinfo
```

Caso a saída do comando não seja vazia, o resultado é positivo.

Há a possibilidade de não utilizar um *hypervisor* para a instalação do Minikube, executando-o ao invés disso sobre o próprio host. Iremos utilizar o Oracle VirtualBox como *hypervisor*, que pode ser encontrado aqui.

Efetue o download e a instalação do Minikube utilizando os seguintes comandos.

```
curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

chmod +x ./minikube

sudo mv ./minikube /usr/local/bin/minikube

minikube version
```

Instalação do Minikube no MacOS

No MacOS, o comando para verificar se o processador suporta virtualização é:

```
sysctl -a | grep -E --color 'machdep.cpu.features|VMX'
```

Se você visualizar `VMX` na saída, o resultado é positivo.

Efetue a instalação do Minikube com um dos dois métodos a seguir, podendo optar-se pelo Homebrew ou pelo método tradicional.

```
sudo brew install minikube

minikube version
```

Ou:

```
curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64

chmod +x ./minikube

sudo mv ./minikube /usr/local/bin/minikube

minikube version
```

Instalação do Minikube no Microsoft Windows

No Microsoft Windows, você deve executar o comando `systeminfo` no prompt de comando ou no terminal. Caso o retorno deste comando seja semelhante com o descrito a seguir, então a virtualização é suportada.

```
Hyper-V Requirements:      VM Monitor Mode Extensions: Yes
                          Virtualization Enabled In Firmware: Yes
                          Second Level Address Translation: Yes
                          Data Execution Prevention Available: Yes
```

Caso a linha a seguir também esteja presente, não é necessária a instalação de um *hypervisor* como o Oracle VirtualBox:

```
Hyper-V Requirements:      A hypervisor has been detected. Features required for
Hyper-V will not be displayed.:      A hypervisor has been detected. Features
required for Hyper-V will not be displayed.
```

Faça o download e a instalação de um *hypervisor* (preferencialmente o Oracle VirtualBox), caso no passo anterior não tenha sido acusada a presença de um. Finalmente, efetue o download do instalador do Minikube aqui e execute-o.

Iniciando, parando e excluindo o Minikube

Quando operando em conjunto com um *hypervisor*, o Minikube cria uma máquina virtual, onde dentro dela estarão todos os componentes do k8s para execução.

É possível selecionar qual *hypervisor* iremos utilizar por padrão, através no comando abaixo:

```
minikube config set driver <SEU_HYPERVISOR>
```

Você deve substituir <SEU_HYPERVISOR> pelo seu hypervisor, por exemplo o KVM2, QEMU, Virtualbox ou o Hyperkit.

Caso não queria configurar um hypervisor padrão, você pode digitar o comando `minikube start --driver=hyperkit` toda vez que criar um novo ambiente.

Certo, e como eu sei que está tudo funcionando como deveria?

Uma vez iniciado, você deve ter uma saída na tela similar à seguinte:

```
minikube start
```

```
😊 minikube v1.26.0 on Debian bookworm/sid
✨ Using the qemu2 (experimental) driver based on user configuration
👍 Starting control plane node minikube in cluster minikube
🔥 Creating qemu2 VM (CPUs=2, Memory=6000MB, Disk=20000MB) ...
🚢 Preparing Kubernetes v1.24.1 on Docker 20.10.16 ...
  ▪ Generating certificates and keys ...
  ▪ Booting up control plane ...
  ▪ Configuring RBAC rules ...
🔍 Verifying Kubernetes components...
  ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
☀️ Enabled addons: default-storageclass, storage-provisioner
🏠 Done! kubectl is now configured to use "minikube" cluster and "default"
namespace by default
```

Você pode então listar os nós que fazem parte do seu *cluster* k8s com o seguinte comando:

```
kubectl get nodes
```

A saída será similar ao conteúdo a seguir:

```
kubectl get nodes
```

Para criar um cluster com mais de um nó, você pode utilizar o comando abaixo, apenas modificando os valores para o desejado:

```
minikube start --nodes 2 -p multinode-cluster
```

```
😊 minikube v1.26.0 on Debian bookworm/sid
✨ Automatically selected the docker driver. Other choices: kvm2, virtualbox,
ssh, none, qemu2 (experimental)
📌 Using Docker driver with root privileges
👍 Starting control plane node minikube in cluster minikube
🚚 Pulling base image ...
💾 Downloading Kubernetes v1.24.1 preload ...
> preloaded-images-k8s-v18-v1...: 405.83 MiB / 405.83 MiB 100.00% 66.78 Mi
> gcr.io/k8s-minikube/kicbase: 385.99 MiB / 386.00 MiB 100.00% 23.63 MiB p
> gcr.io/k8s-minikube/kicbase: 0 B [_____] ?% ? p/s 11s
```

```
🔥 Creating docker container (CPUs=2, Memory=8000MB) ...
🐳 Preparing Kubernetes v1.24.1 on Docker 20.10.17 ...
  ▪ Generating certificates and keys ...
  ▪ Booting up control plane ...
  ▪ Configuring RBAC rules ...
🔗 Configuring CNI (Container Networking Interface) ...
🔍 Verifying Kubernetes components...
  ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
☀️ Enabled addons: storage-provisioner, default-storageclass

👍 Starting worker node minikube-m02 in cluster minikube
🚚 Pulling base image ...
🔥 Creating docker container (CPUs=2, Memory=8000MB) ...
🌐 Found network options:
  ▪ NO_PROXY=192.168.11.11
🐳 Preparing Kubernetes v1.24.1 on Docker 20.10.17 ...
  ▪ env NO_PROXY=192.168.11.11
🔍 Verifying Kubernetes components...
🏁 Done! kubectl is now configured to use "minikube" cluster and "default"
namespace by default
```

Para visualizar os nós do seu novo cluster Kubernetes, digite:

```
kubectl get nodes
```

Inicialmente, a intenção do Minikube é executar o k8s em apenas um nó, porém a partir da versão 1.10.1 é possível usar a função de multi-node.

Caso os comandos anteriores tenham sido executados sem erro, a instalação do Minikube terá sido realizada com sucesso.

Ver detalhes sobre o cluster

```
minikube status
```

Descobrindo o endereço do Minikube

Como dito anteriormente, o Minikube irá criar uma máquina virtual, assim como o ambiente para a execução do k8s localmente. Ele também irá configurar o `kubectl` para comunicar-se com o Minikube. Para saber qual é o endereço IP dessa máquina virtual, pode-se executar:

```
minikube ip
```

O endereço apresentado é que deve ser utilizado para comunicação com o k8s.

Acessando a máquina do Minikube via SSH

Para acessar a máquina virtual criada pelo Minikube, pode-se executar:

```
minikube ssh
```

Dashboard do Minikube

O Minikube vem com um *dashboard web* interessante para que o usuário iniciante observe como funcionam os *workloads* sobre o k8s. Para habilitá-lo, o usuário pode digitar:

```
minikube dashboard
```

Logs do Minikube

Os *logs* do Minikube podem ser acessados através do seguinte comando.

```
minikube logs
```

Remover o cluster

```
minikube delete
```

Caso queira remover o cluster e todos os arquivos referente a ele, utilize o parametro *--purge*, conforme abaixo:

```
minikube delete --purge
```

Kind

O Kind (*Kubernetes in Docker*) é outra alternativa para executar o Kubernetes num ambiente local para testes e aprendizado, mas não é recomendado para uso em produção.

Instalação no GNU/Linux

Para fazer a instalação no GNU/Linux, execute os seguintes comandos.

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.14.0/kind-linux-amd64  
  
chmod +x ./kind  
  
sudo mv ./kind /usr/local/bin/kind
```

Instalação no MacOS

Para fazer a instalação no MacOS, execute o seguinte comando.

```
sudo brew install kind
```

ou

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.14.0/kind-darwin-amd64  
chmod +x ./kind  
mv ./kind /usr/bin/kind
```

Instalação no Windows

Para fazer a instalação no Windows, execute os seguintes comandos.

```
curl.exe -Lo kind-windows-amd64.exe  
https://kind.sigs.k8s.io/dl/v0.14.0/kind-windows-amd64  
  
Move-Item .\kind-windows-amd64.exe c:\kind.exe
```


Instalação no Windows via Chocolatey

Execute o seguinte comando para instalar o Kind no Windows usando o Chocolatey.







```
choco install kind
```

Criando um cluster com o Kind

Após realizar a instalação do Kind, vamos iniciar o nosso cluster.

```
kind create cluster
```

```
Creating cluster "kind" ...
```

- ✓ Ensuring node image (kindest/node:v1.24.0) 
- ✓ Preparing nodes 
- ✓ Writing configuration 
- ✓ Starting control-plane 
- ✓ Installing CNI 
- ✓ Installing StorageClass 

```
Set kubectl context to "kind-kind"
```

```
You can now use your cluster with:
```







```
kubectl cluster-info --context kind-kind
```

Not sure what to do next? 😊 Check out
<https://kind.sigs.k8s.io/docs/user/quick-start/>

É possível criar mais de um cluster e personalizar o seu nome.

```
kind create cluster --name giropops
```

```
Creating cluster "giropops" ...
```

- ✓ Ensuring node image (kindest/node:v1.24.0) 
- ✓ Preparing nodes 
- ✓ Writing configuration 
- ✓ Starting control-plane 
- ✓ Installing CNI 
- ✓ Installing StorageClass 

```
Set kubectl context to "kind-giropops"
```

```
You can now use your cluster with:
```

```
kubectl cluster-info --context kind-giropops
```

Thanks for using kind! 😊

Para visualizar os seus clusters utilizando o kind, execute o comando a seguir.

```
kind get clusters
```

Liste os nodes do cluster.

```
kubectl get nodes
```

Criando um cluster com múltiplos nós locais com o Kind

É possível para essa aula incluir múltiplos nós na estrutura do Kind, que foi mencionado anteriormente.

Execute o comando a seguir para selecionar e remover todos os clusters locais criados no Kind.

```
kind delete clusters $(kind get clusters)
```










```
Deleted clusters: ["giropops" "kind"]
```

Crie um arquivo de configuração para definir quantos e o tipo de nós no cluster que você deseja. No exemplo a seguir, será criado o arquivo de configuração `kind-3nodes.yaml` para especificar um cluster com 1 nó control-plane (que executará o control plane) e 2 workers.

```
cat << EOF > $HOME/kind-3nodes.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
  - role: worker
EOF
```

Agora vamos criar um cluster chamado `kind-multinodes` utilizando as especificações definidas no arquivo `kind-3nodes.yaml`.

```
kind create cluster --name kind-multinodes --config $HOME/kind-3nodes.yaml
```

```
Creating cluster "kind-multinodes" ...
✓ Ensuring node image (kindest/node:v1.24.0) 
✓ Preparing nodes   
✓ Writing configuration 
✓ Starting control-plane 
✓ Installing CNI 
✓ Installing StorageClass 
✓ Joining worker nodes 
Set kubectl context to "kind-kind-multinodes"
You can now use your cluster with:
```

```
kubectl cluster-info --context kind-kind-multinodes
```

Have a question, bug, or feature request? Let us know!
<https://kind.sigs.k8s.io/#community> 

Valide a criação do cluster com o comando a seguir.

```
kubectl get nodes
```

Mais informações sobre o Kind estão disponíveis em: <https://kind.sigs.k8s.io>

Primeiros passos no k8s

Verificando os namespaces e pods

O k8s organiza tudo dentro de *namespaces*. Por meio deles, podem ser realizadas limitações de segurança e de recursos dentro do *cluster*, tais como *pods*, *replication controllers* e diversos outros. Para visualizar os *namespaces* disponíveis no *cluster*, digite:

```
kubectl get namespaces
```

Vamos listar os *pods* do *namespace* kube-system utilizando o comando a seguir.

```
kubectl get pod -n kube-system
```

Será que há algum *pod* escondido em algum *namespace*? É possível listar todos os *pods* de todos os *namespaces* com o comando a seguir.

```
kubectl get pods -A
```

Há a possibilidade ainda, de utilizar o comando com a opção `-o wide`, que disponibiliza maiores informações sobre o recurso, inclusive em qual nó o *pod* está sendo executado. Exemplo:

```
kubectl get pods -A -o wide
```

Executando nosso primeiro pod no k8s

Iremos iniciar o nosso primeiro *pod* no k8s. Para isso, executaremos o comando a seguir.

```
kubectl run nginx --image nginx
```

```
pod/nginx created
```

Listando os *pods* com `kubectl get pods`, obteremos a seguinte saída.

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	66s

Vamos agora remover o nosso *pod* com o seguinte comando.

```
kubectl delete pod nginx
```

A saída deve ser algo como:

```
pod "nginx" deleted
```

Executando nosso primeiro pod no k8s

Uma outra forma de criar um pod ou qualquer outro objeto no Kubernetes é através da utilização de uma arquivo manifesto, que é uma arquivo em formato YAML onde você passa todas as definições do seu objeto. Mas pra frente vamos falar muito mais sobre como construir arquivos manifesto, mas agora eu quero que você conheça a opção `--dry-run` do `kubectl`, pois com ele podemos simular a criação de um resource e ainda ter um manifesto criado automaticamente.

Exemplos:

Para a criação do template de um *pod*:

```
kubectl run meu-nginx --image nginx --dry-run=client -o yaml >
pod-template.yaml
```

Aqui estamos utilizando ainda o parametro `'-o'`, utilizando para modificar a saída para o formato YAML.

Para a criação do *template* de um *deployment*:

Com o arquivo gerado em mãos, agora você consegue criar um pod utilizando o manifesto que criamos da seguinte forma:

```
kubectl apply -f pod-template.yaml
```

Não se preocupe por enquanto com o parametro `'apply'`, nós ainda vamos falar com mais detalhes sobre ele, nesse momento o importante é você saber que ele é utilizado para criar novos resources através de arquivos manifestos.

Expondo o pod e criando um Service

Dispositivos fora do *cluster*, por padrão, não conseguem acessar os *pods* criados, como é comum em outros sistemas de contêineres. Para expor um *pod*, execute o comando a seguir.

```
kubectl expose pod nginx
```

Será apresentada a seguinte mensagem de erro:

```
error: couldn't find port via --port flag or introspection
See 'kubectl expose -h' for help and examples
```

O erro ocorre devido ao fato do k8s não saber qual é a porta de destino do contêiner que deve ser exposta (no caso, a 80/TCP). Para configurá-la, vamos primeiramente remover o nosso *pod* antigo:

```
kubectl delete -f pod-template.yaml
```

Agora vamos executar novamente o comando para a criação do pod utilizando o parametro 'dry-run', porém agora vamos adicionar o parametro '--port' para dizer qual a porta que o container está escutando, lembrando que estamos utilizando o nginx nesse exemplo, um webserver que escuta por padrão na porta 80.

```
kubectl run meu-nginx --image nginx --dry-run=client -o yaml >
pod-template.yaml
kubectl create -f pod-template.yaml
```

Liste os pods.

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
meu-nginx	1/1	Running	0	32s

O comando a seguir cria um objeto do k8s chamado de *Service*, que é utilizado justamente para expor *pods* para acesso externo.

```
kubectl expose pod meu-nginx
```

Podemos listar todos os *services* com o comando a seguir.

```
kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	8d
nginx	ClusterIP	10.105.41.192	<none>	80/TCP	2m30s

Como é possível observar, há dois *services* no nosso *cluster*: o primeiro é para uso do próprio k8s, enquanto o segundo foi o quê acabamos de criar.

Limpendo tudo e indo para casa

Para mostrar todos os recursos recém criados, pode-se utilizar uma das seguintes opções a seguir.

```
kubectl get all
```

```
kubectl get pod,service
```

```
kubectl get pod,svc
```

Note que o k8s nos disponibiliza algumas abreviações de seus recursos. Com o tempo você irá se familiarizar com elas. Para apagar os recursos criados, você pode executar os seguintes comandos.

```
kubectl delete -f pod-template.yaml
```

```
kubectl delete service nginx
```

Liste novamente os recursos para ver se os mesmos ainda estão presentes.