Introdução ao Kubernetes

Mestrando: João Ernesto Arzamendia - MPCA - Facom
Orientador: Dionisio Machado Leite Filho - MPCA - Facom

Abril de 2020

Conteúdo

Lista de Figuras

1 Introdução

Kubernetes[1] é o Orquestrador de Containers mais utilizado atualmente. Ele é considerado o padrão de mercado quando se trata de hospedagem de aplicações em containers, principalmente na nuvem. Seu uso torna possível garantir alta disponibilidade, *self-healing*[2] e escalabilidade de forma simples e eficaz a diversos tipos de aplicações.

Orquestradores de Containers em geral, trazem diversos benefícios para aplicações empacotadas em containers. Entre os principais benefícios, Asif Khan [3] lista:

1. Gerenciamento de estado de Cluster;
2. Alta disponibilidade e tolerância a falhas;
3. Segurança;
4. Simplificação de rede;
5. Descoberta de serviço;
6. Implantação Contínua (CI);
7. Monitoramento e governança.

Aplicações em containers tem todas as suas bibliotecas de software e sistema operacional empacotadas em um "único" arquivo que pode ser executado e replicado em qualquer ambiente que execute containers. Isto torna a infraestrutura necessária para a hospedagem de aplicações muito mais padronizada e reprodutível.

Para compreender melhor as vantagens e desvantagens que este modelo pode trazer para o ambiente de hospedagem e desenvolvimento de aplicações, precisamos adentrar em alguns conceitos importantes.

1.1 Containers

É muito comum falar de Docker[4] quando estamos falando sobre Containers. De fato, a popularização atual do uso de containers foi muito ajudada pelas

[1] site do Kubernetes: https://kubernetes.io/

[2] Self-healing é a capacidade de se recuperar de erros de forma automática.

[3] Key Characteristics of a Container Orchestration Platform to Enable a Modern Application, 2017. DOI: 10.1109/MCC.2017.4250933

[4] Site do Docker: https://www.docker.com/

facilidades que a plataforma de gerenciamento de containers Docker entregou para o mercado. Porém o termo "Container" é muito mais antigo do que o Docker, que foi disponibilizado ao público por volta de 2013.

O que chamamos atualmente de Containers pode ser mais precisamente descrito como "Virtualização em nível de sistema operacional". Máquinas Virtuais convencionais , em geral, utilizam um software chamado de Hypervisor[5] para criar uma camada de abstração entre o hardware de um servidor físico e múltiplos sistemas operacionais virtuais isolados.

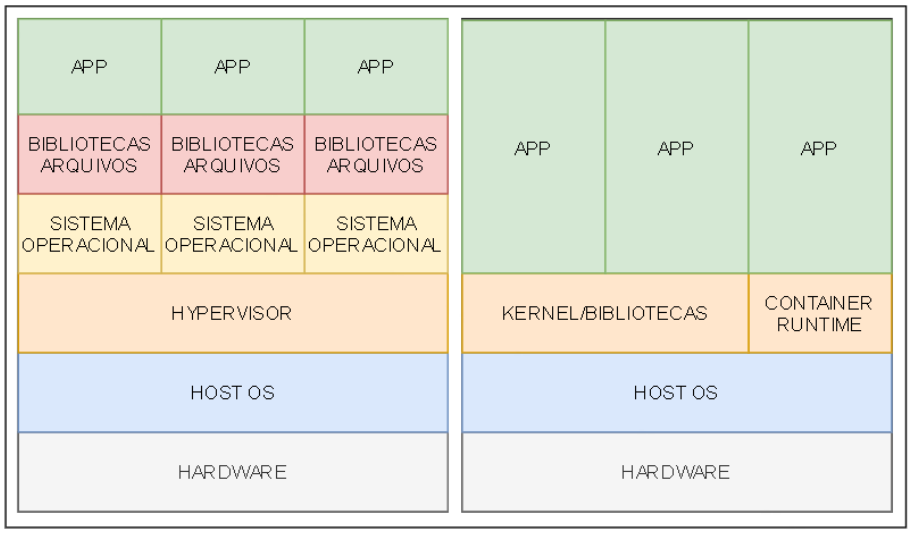

Figura 1: Diferenças entre Virtualização convencional e virtualização baseada em containers.

Já Containers, utilizam um software chamado Container Engine e diversos recursos de Kernel[6] para compartilhar os recursos do servidor hospedeiro entre diversos processos isolados que simulam sistemas operacionais completos. Com isto temos uma grande economia em recursos, já que todos os containers no mesmo servidor compartilharam o mesmo Kernel e Sistema Operacional. Na figura 1 podemos ver as diferenças entre ambas as abordagens.

A origem dos containers pode remontar à década de 1950 com a introdução do conceito de multiprogramação. Conforme Randal [7] descreve, diversos trabalhos tratam sobre o tema, diversas novas tecnologias nasceram, morreram ou foram adaptadas até chegarmos ao conceito moderno de containers.

[5] https://en.wikipedia.org/wiki/Hypervisor

[6] Kernel é o núcleo de um Sistema Operacional, ele oferece os serviços básicos necessários para o seu funcionamento.

[7] The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers, 2017. DOI: 10.1145/3365199

Podemos seguir o seguinte fluxo para compreender a atual implementação dos containers modernos: Nos anos 2000, a FreeBSD lançou o Jails[8] que através do uso recursos como o Chroot, conseguia isolar processos em ambientes controlados, ou "celas", como a tradução nos revela.

Continuando para 2001, o Linux VServer[9] nos apresentou avanços no isolamento em nível de kernel, tornando possível que múltiplos processo isolados compartilhassem com segurança um mesmo Kernel de sistema operacional.

No mesmo caminho, o OpenVZ[10] trouxe melhorias no Kernel do Linux, adicionando mais funcionalidades e tornando o isolamento de processos mais abrangente. Com isto, dois grandes projetos foram iniciados: Em 2008 foi criado o Linux Containers[11], também chamado de LXC e em 2013 o projeto Docker.

O LXC combinou uma série de tecnologias criadas anteriormente em uma poderosa ferramenta de criação e inicialização de containers nativa do Linux, que pode ser usada até hoje. Em seu site, o LXC lista os seguintes recursos de Kernel usados no isolamento de processos:

- Kernel namespaces;
- Apparmor and SELinux;
- Seccomp policies;
- Chroots;
- Kernel capabilities;
- CGroups.

O projeto Docker, inicialmente foi criado como uma plataforma de gerenciamento de containers baseada no LXC. Com sua interface de linha de comando extremamente simples e intuitiva era possível criar containers a partir de arquivos de texto, iniciar sua execução e monitorar e gerenciar os containers em execução.

Além disso, duas novas tecnologias foram incorporadas em sua plataforma: O Union Filesystem e os Registros de Containers.

[8] https://wiki.freebsd.org/Jails
[9] http://linux-vserver.org/
[10] https://openvz.org/
[11] https://linuxcontainers.org/

1.1.1 Union Filesystem

Com o Union Filesystem[12], o sistema de arquivos pode ser separado em camadas, chamadas de branchs, essas camadas podem ser sobrepostas e compartilhadas de forma transparente, criando no fim um sistema de arquivos único e coerente. Com esta implementação é possível economizar espaço em disco além de otimizar o download de camadas quando repetidas.

Essas camadas são então logicamente empilhadas e empacotadas em uma imagem de container. Virtualmente, o que vemos é um pacote fechado, com toda a aplicação e suas dependências em um único arquivo. Na realidade, caso alguma dessas camadas seja igual a de outra imagem de container, apenas uma existirá de fato.

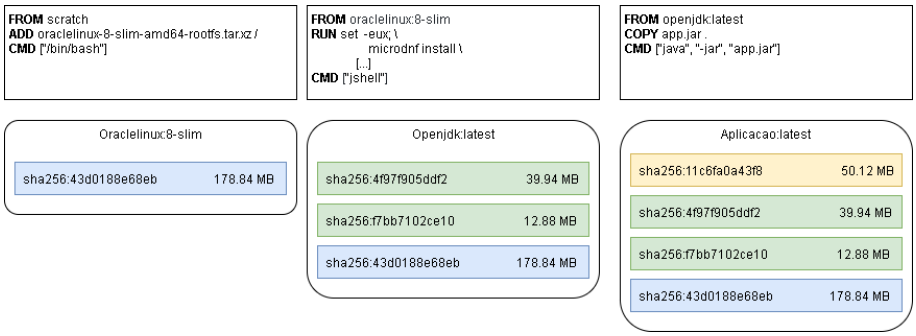

Figura 2: O conjunto de camadas que criam uma aplicação baseada na imagem Openjdk:latest.

Outro ponto muito importante é que as camadas, uma vez criadas, não podem mais ser editadas. Apenas a última camada sobreposta sobre a pilha é aberta para escrita. Essa última camada é criada automaticamente com a inicialização do container, ao final de sua execução ela é descartada. Em geral, é calculado um hash usando o algorítimo SHA256 para cada camada, tornando possível a rápida verificação de camadas repetidas.

Com isto, nós temos uma imagem de container imutável, que sempre será iniciada de um mesmo ponto, definido em seu sistema de arquivos. Unindo isso à capacidade de compartilhamento de camadas, as imagens de containers se tornaram um método extremamente popular de compartilhamento de aplicações. Porém para esta popularização, a próxima tecnologia que abordaremos foi fundamental.

[12]O Docker atualmente utiliza o AUFS e o OverlayFS como storage drivers. Mais informações em: https://docs.docker.com/storage/storagedriver/select-storage-driver/

1.1.2 Registros de Containers

Imagens de containers podem ter suas camadas exportadas na forma de arquivos binários[13] compactados. Estes arquivos podem ser compartilhados através de mídias removíveis ou serviços de compartilhamento de arquivos, para então serem incorporados no Union FileSystem e iniciadas na forma de Containers.

Este processo ignora uma das principais vantagens que o UnionFS apresenta: o compartilhamento de camadas. Para resolver isto, os Registros de Containers foram criados. O Registro de Containers é um servidor de compartilhamento de imagens de containers que as disponibiliza em camadas. Caso uma das camadas já se encontre no cliente que tenta fazer download, apenas as camadas que faltam são baixadas.

Utilizando essa tecnologia, a Docker criou o DockerHub[14]. Com ele empresas e pessoas podem fazer o upload de suas imagens em uma plataforma gratuita de hospedagem de imagens. Além disso, eles criaram o conceito de imagens base.

Imagens base são imagens de contêineres especialmente criadas para serem utilizadas como base de criação de outras imagens. Elas contém dependências comumente usadas para diversas aplicações e são extremamente otimizadas em relação a tamanho e usabilidade. Na tabela 1 podemos ver algumas delas.

Nome	Imagem base	Tamanho em disco
alpine:3.15	scratch	5.59 MB
busybox:1.34.1	scratch	1.24 MB
centos:7	scratch	204 MB
openjdk:19-jdk-alpine3.15	alpine:3.15	333MB

Table 1: Imagens base de containers, suas próprias imagens bases e tamanho em disco.

Existem diversos Registros de Containers disponíveis no mercado. Desde versões gerenciadas por provedores de computação em nuvem, onde você paga pelo o que usar, até versões de código aberto que podem ser instalados localmente em nossos ambientes.

[13] Blobs - Binary large object
[14] https://hub.docker.com/

1.1.3 Dockerfiles

Uma vez que vimos para que servem e onde vivem as Imagens de Containers, precisamos discutir como elas são criadas. Imagens de Containers Docker são criadas através de Dockerfiles[15].

Dockerfiles, conforme podemos ver no código 1.1.3, são arquivos de texto com instruções de como a Imagem deve ser construída. Comandos podem ser executados, arquivos adicionados e outras imagens de container referenciadas. Na tabela 2 temos uma lista dos principais comandos.

No caso do Dockerfile, toda linha deve ser iniciada com um comando, como RUN para executar um comando no sistema operacional do container, COPY para adicionar um arquivo ou USER para definir que usuário irá ser executado pelos processos da imagem. Um comando muito importante a ser observado é o FROM.

O comando FROM define qual imagem base será usada. Com isto, todas as dependências instaladas na imagem referenciada estarão a disposição da imagem que está sendo construída. É comumente estabelecido que toda Dockerfile comece com o comando FROM.

Código 1: Exemplo de Dockerfile que criará a imagem de um simples servidor Web baseado em Nginx.

```
1 FROM nginx:alpine
2 WORKDIR /usr/share/nginx/html
3 RUN echo "<html><h3>Facom Rulez!</h3></html>" > index.html
4 EXPOSE 80
```

Por convenção, os Dockerfiles devem ter seu nome de arquivo como "Dockerfile", sem extensão. Assim a interface de linha de comando identifica o arquivo com o nome de Dockerfile no diretório de execução do comando[16] e iniciará a construção da imagem. Também é possível utilizar o argumento "-f" para definir o caminho onde o Dockerfile se encontra.

1.1.4 O Ciclo de vida de um container

Containers respeitam um ciclo de vida muito simples. Containers basicamente são iniciados, executam um processo e ao fim dele, morrem sem deixar vestígios

[15] https://docs.docker.com/engine/reference/builder/

[16] O comando usado para construir uma imagem é o: "docker build -t nomedaimagem .", onde . define o diretório de execução do comando

Comando	Descrição
FROM	Define a imagem base do container.
COPY	Copia um arquivo do filesystem local para o container.
RUN	Executa um ou mais comandos no container.
WORKDIR	Define o diretório de execução atual.
USER	Define o usuário executará os processos do container.
ENTRYPOINT	Define o processo principal do container.

Table 2: Principais comandos disponíveis para uso nas Dockerfiles.

de sua existência.

O processo executado por um container é definido pelo comando ENTRYPOINT. O container se manterá em execução enquanto este processo continuar sendo executado. Por mais que processos filhos ainda estejam sendo executados, o processo principal é o que define o estado do container.

Como discorremos antes, todas as modificações feitas em disco na última camada do Union Filesystem do container são descartadas ao final de sua execução. Por isto é muito comum definirmos que containers são objetos descartáveis. Processos de atualização, reinicialização e correção de erros na maioria das vezes envolvem a destruição de um container e a criação de um container totalmente novo.

1.2 Gado x Pet

Um conceito interessante criada e a analodiga do Gado versus Pet. Ela foi criada por Bill Baker[17] na descrição do processo de escalamento horizontal e vertical de servidores de banco de dados SQL.

Esta analogia se aplica muito bem ao universo de containers e servidores tradicionais. Temos que em geral, os servidores tradicionais são tratados como pets. Servidores tem um nomes que podem ser memorizados. Quando um servidor apresenta algum problema, um administrador de sistemas será definido para trata-lo. Caso nosso servidor não consiga atender a demanda de trabalho a ele determinada, podemos aumentar seus recursos ou aplicar atualizações.

Já com containers, temos um caso diferente. Containers em geral não tem nomes, eles recebem códigos gerados automaticamente. Quando um container apresenta problemas, ele é destruído e outro é colocado no seu lugar, muitas vezes de forma automática. Por fim, aplicações hospedadas em containers, quando não conseguem atender a demanda, em geral são escaladas horizontalmente, isto é, adicionando mais containers para responder a carga de trabalho

[17] http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/

demandada.

Vemos com isso, que containers se assemelham muito mais a gados do que a pets. Sendo assim, é imporante ter isso em mente durante o gerenciamento de aplicações hospedadas com containers. Esses são ambientes muito mais efêmeros do que servidores convencionais ou maquinas virtuais.

Para finalizarmos, ao utilizar containers na hospedagem de aplicações em ambientes de produção, uma série de dificuldades começam a aparecer. Como podemos controlar a atualização de dezenas de containers de um determinado sistema distribuído em diversos servidores? Como garantir que falhas sejam automaticamente encontradas e solucionadas? Essas e outras questões podem ser resolvidas com o uso de Orquestradores de Containers.

1.3 Orquestradores de Containers

Orquestradores de Containers são responsáveis pela implementação, monitoramento e controle de containers em ambientes de múltiplos servidores. Eles são essenciais para ambientes empresariais de hospedagem de aplicações.

Emiliano Casalicchio[18] discorre que "Orquestração de Containers permite que provedores de nuvens e aplicações definem como selecionar, implantar, monitorar e controlar dinamicamente a configuração de aplicativos multi-container na nuvem."

Hoje existem diversos Orquestradores de Containers disponíveis tanto na nuvem quanto para instalações on-premises, ou locais. Entre os principais estão: Kubernetes, Docker Swarm, Openshift e Mesos.

1.3.1 Orquestração

Orquestrar containers é uma função muito importante em ambientes de hospedagem de aplicações.

Pense no papel de um Orquestrador de Containers como o de um engenheiro civil em um grande canteiro de obras. No canteiro de obras cada trabalhador tem seu papel definido, porém se cada um tentar fazer seu trabalho da maneira que achar melhor, a chance do projeto dar certo é extremamente baixa.

O Engenheiro então, observando um projeto definido anteriormente, supervisiona o trabalho dos encarregados de cada função, gerencia os recursos

[18] Autonomic Orchestration of Containers: Problem Definition and Research Challenges, 2047. DOI: 10.4108/eai.25-10-2016.2266649

disponíveis para a obra, faz relatórios de desempenho e quando necessário faz novas contratações.

No caso dos Orquestradores de Containers, este projeto é em geral definido de maneira declarativa e é chamado de Estado do Cluster. Basicamente nós decidimos qual é o estado desejado do ambiente e o Orquestrador irá monitorar todo o ambiente fazendo modificações para alcançar o estado desejado.

1.3.2 Kubernetes

O Orquestrador de Containers mais utilizado no mercado atualmente é o Kubernetes. Conforme descrito em seu site institucional[19]: Kubernetes (K8s) é um produto Open Source utilizado para automatizar a implantação, o dimensionamento e o gerenciamento de aplicativos em contêiner.

Entre as vantagens no uso do Kubernetes como Orquestrador de Containers está o fato deste ter uma das maiores comunidades de código aberto que oferece suporte gratuito e com uma grande base de conhecimento disponível em diversos sites da internet.

O Kubernetes, assim como os Containers, nasceu através da evolução de uma serie de tecnologias e ferramentas mais antigas, estando em constante evolução desde sua criação. A Google, conforme BURNS Et Al[20] descreve que, muito cedo começou a trabalhar com Containers Linux e para isto criou uma série de sistemas de gerenciamento de containers em alta escala.

Com isto, surgiu o Borg [21], um sistema de gerenciamento de containers hospedados internamente pela Google. Ele foi fortemente influenciado por sistemas que gerenciam separadamente a execução de tarefas e processos de longa duração, facilitando o gerenciamento de containers para as diversas equipes da Google.

O uso do Borg foi um sucesso e com sua disseminação entre os engenheiros do Google, rapidamente a necessidade de unificação de processos e ferramentas fez com que a plataforma Omega fosse criada. Como sucessora do projeto Borg, o projeto Omega adicionou um banco centralizado para o estado do cluster baseado no algorítimo Paxos[22]. Com isto, foi possível sair de uma abordagem monolítica para uma de múltiplos processos acessando uma mesma base baseada em consenso.

[19] https://kubernetes.io/pt-br/

[20] Lessons learned from three containermanagement systems over a decade, 2016. ACM Queue, vol. 14

[21] Large-scale cluster management at Google with Borg, 2015

[22] Paxos é um protocolo de criação de consenso entre processos diferentes em ambiente de clusters.

Com a popularização do uso de Containers e dos esforços da Google como provedora de infraestrutura de Computação em Nuvem, surge o projeto Kubernetes, que diferente de seus predecessores foi concebido com um projeto de código livre extremamente focado na experiência de desenvolvedores criando aplicações para rodar em clusters.

Por fim, o projeto Kubernetes foi doado para a Cloud Native Computing Foundation[23], fundação que abarca uma serie de grandes projetos de código aberto voltados a Computação em Nuvem e não para de crescer seja em número de usuários ou em linhas de código em seu projeto. Caso você tenha interesse, acesse https://github.com/kubernetes/kubernetes e verifique como contribuir com o projeto.

1.4 Conclusão

Neste capítulo nós discutimos os conceitos sobre Containers, Orquestradores de Containers e as tecnologias que tornaram possível a popularização do Docker e do Kubernetes.

Fora os pontos mencionados até agora, diversos outros projetos, sejam de código aberto ou não, ajudaram na popularização da tecnologia de Containers. O eco-sistema atual é gigantesco e não para de crescer.

Grupos como a CNCF ajudam a manter o Kubernetes, que hoje é considerado o habitat natural de aplicações feitas para rodar em escala e/ou usando Computação em Nuvem.

Nos próximos capítulos vamos abordar melhor os recursos básicos de uma aplicação em um Cluster Kubernetes.

[23] https://www.cncf.io/

2 Componentes do Kubernetes

O Kubernetes é formado por uma série de componentes que compartilham um mesmo estado em um banco de dados do tipo Chave-Valor, o ETCD. A interação entre esses diversos componentes gerenciam um cluster que pode ser formado por centenas ou até mesmo milhares de containers e servidores.

Podemos dividir esses componentes em 3 camadas: Os componentes do Control Plane, os componentes do Node Plane e os Addons.

O Control Plane é responsável pelas decisões administrativas do ambiente. Pense nele como a cabeça por trás das decisões tomadas pelo Kubernetes.

No Node Plane são reunidos os componentes que são executados em todos os servidores do Cluster. Nele temos os componentes que gerenciam os container e rede dos servidores.

Por fim, os Addons são componentes que usam recursos do Kubernetes para adicionar capacidades extras ao Cluster.

Neste capítulo abordaremos os principais componentes do Kubernetes, sua utilidade e uso.

2.1 Kube-apiserver

O Kube-apiserver é o componente que dá acesso ao Control Plane e é a porta de entrada do cluster. Todo acesso ao cluster é mediado por esta API. Quando executamos um comando através do executável kubectl, estamos fazendo uma chamada a API do Kubernetes, que está sendo mantida pelo kube-apiserver.

A segurança e disponibilidade da API do Kubernetes é de extrema importância para o ambiente, por isto, em geral ela é executada em servidores separados, comumente chamados de Master Nodes.

Pense na API do Kubernetes como um ponto de acesso. Outros recursos a acessam de maneira padronizada, tornando possível controlar o Kubernetes de diversas maneiras. Seja através da tela preta de uma linha de comando ou através de Dashboards web extremamente intuitivos e customizáveis.

2.2 ETCD

Todos os dados do Kubernetes são mantidos no ETCD, que é um banco de dados do tipo chave-valor baseado em consenso. Nele o Estado do Cluster é mantido

e por isso o ETCD prioriza a consistência dos dados sobre disponibilidade.

É possível manter várias réplicas do ETCD ativas ao mesmo tempo, e mesmo a falha de uma delas não afetará a disponibilidade do ambiente e integridade dos dados do Kubernetes. Isso é possível pelo algorítimo de consenso usado pelo ETCD para manter seu estado.

O ETCD foi criado pela CoreOS em 2013 e posteriormente (assim como o Kubernetes) foi doado para a CNCF como um projeto de código aberto. Com isto, ele é hoje também usado por muitos outros projetos como um banco de dados para estados de clusters.

2.3 Kube-scheduler

O Kube-scheduler é o responsável por determinar em que servidor cada pod será iniciado e executado. Para tomar esta decisão ele leva em conta diversas questões, como disponibilidade de recursos, políticas aplicadas às cargas de trabalho e atribuições especiais vindas do usuário.

Ele verifica os recursos e peculiaridades que podem ser atribuídas para cada um dos servidores do cluster e com base na especificação de cada Pod, ele decide em que servidor esse pode deve ser iniciado. É importante notar também, que é atribuição do kube-scheduler decidir quando um Pod deve ser removido de um servidor, de forma voluntária ou não.

2.4 Kube-controller-manager

O Kube-controller-manager é responsável por gerir as tarefas administrativas do cluster. Nele, diversos Controladores com diversas responsabilidades distintas são executados a fim de manter o estado do cluster atualizado.

Ele é responsável, entre outras coisas, por gerir e solicitar a execução de jobs na forma de Pods, criar os tokens de acesso a API para recursos do Kubernetes e gerenciar o estado dos servidores do cluster.

2.4.1 Endpoints Controller

O Endpoint Controller é responsável por adicionar Endpoints, ou IP's de Pods, a lista de Endpoint de um Service. Ele observa as regras definidas no campo Selector de um Service para descobrir que Pods devem ser adicionados a lista, tornando possível a descoberta de Serviço de forma fácil e prática usando DNS e Services.

2.4.2 Cloud Controller

O Cloud Controller é um Controlador especial que faz o meio de campo entre o Kubernetes e a Cloud onde ele está hospedado. Cada versão gerenciada tem uma versão específica do Cloud Controller Manager, fazendo com que a interação entre o Kubernetes e a Cloud específica aconteça da melhor maneira possível.

Com ele, os Services do Kubernetes conseguem adquirir IP's públicos válidos, o Kubernetes consegue gerenciar discos virtuais de forma automática.

2.5 Kubelet

O Kubelet é o processo responsável por iniciar os containers definidos nos pods. Ele conversa com o Container Runtime instalado no servidor garantindo a execução e saúde dos containers definidos no Estado do Cluster.

É importante notar que o Kubelet só controla os containers criados a partir de Pods. Em geral, a maioria dos Container Runtimes podem ser usados pelo Kubelet. Para que isto seja possível, o Container Runtime deve dar suporte ao Container Runtime Interface (CRI), que é uma interface padrão usada pelo Kubernetes na conexão Kubelet e Container Runtime.

2.5.1 Docker Shim

2.6 Kube-proxy

O Kube-proxy é responsável por gerenciar a rede de comunicação entre os pods e serviços do Kubernetes. Ele cria regras de rede para garantir que a comunicação entre Services[24] e Pods do cluster ocorra de maneira transparente, independente das possíveis trocas de pods que possam acontecer.

Ele é executado em todos os servidores do cluster, em geral na forma de um Pod de alta prioridade.

2.7 kubectl

O Kubectl é a forma mais básica de se gerenciar um cluster Kubernetes. Através da combinação de seus comandos é possível gerenciar e monitorar todo o cluster.

[24] Service: Um recurso do Kubernetes.

Ele é um executável extremamente leve disponível para a maioria dos sistemas operacionais e pode controlar clusters locais ou remotos.

Sua instalação é simples e pode ser encontrada em: https://kubernetes.io/docs/tasks/tools/.

2.8 Conclusão

Neste capítulo abordamos os principais recursos do Kubernetes. Vários recursos não foram abordados nesta introdução, porém serão vistos mais à frente. Já outros, seja por falta de espaço ou por serem usados em casos muito específicos, não serão abordados neste documento.

É importante ter um conhecimento genérico dos principais componentes e posteriormente ir conhecendo um a um todos os outros, o que é uma tarefa árdua, visto que novos recursos são lançados com frequência. Além disso, recursos customizados da comunidade são tão úteis que acabam se confundindo com recursos do próprio Kubernetes, quando não, são até mesmo incorporados realmente.

3 Criando cluster Kubernetes com o Kind

Como vimos anteriormente, um cluster Kubernetes é formado por diversos componentes que trabalham em conjunto de forma distribuída para gerenciar e implantar o estado do cluster. Existem várias formas de se criar um cluster Kubernetes.

Para facilitar os testes e atividades propostas neste documento, utilizaremos um cluster de Kubernetes local criado através da ferramenta Kind[25], que utiliza o Docker para criar um cluster Kubernetes para desenvolvimento local.

3.1 Requisitos

Os seguintes recursos serão utilizados no decorrer deste documento:

- Powershell 5
- Docker for Windows 4.1.1
- Kind 0.11.1
- Git 2.33.0.windows.1

3.2 Powershell

Todos os comandos usados neste documento foram executados utilizando um computador com o Windows 11, usando o Powershell 5 como interpretador. É importante notar que caso os comandos sejam executados em outro interpretador, como o BASH ou CMD, algumas modificações poderão ser necessárias.

3.3 Docker

A versão do Docker utilizada foi a Docker for Windows 4.1.1, em sua versão gratuita. O computador utilizado da suporte ao WSL v2 e essa função fui utilizada. A maioria dos recursos utilizados devem funcionar em versões diferentes do Docker, incluindo a versão OpenSource do Docker para Linux.

[25] https://kind.sigs.k8s.io/

3.4 Kind

O Kind é uma ferramenta de criação de clusters locais do Kubernetes usando Docker. Ele é disponível para diversos sistemas operacionais, se tratando apenas de um executável disponível no site https://kind.sigs.k8s.io/.

3.5 Git

O Git for Windows versão 2.33.0 foi utilizado para salvar os arquivos utilizados neste documento. Todos os arquivos usados neste documento estão versionados no github https://github.com/Jarzamendia/apostila.

3.6 Criando Clusters

Após fazer download da última versão do Kind em seu repositório[26] e adiciona-lo ao Path do Windows, é possível criar clusters de desenvolvimento utilizando o comando 'Kind create cluster', conforme vemos na figura 3.

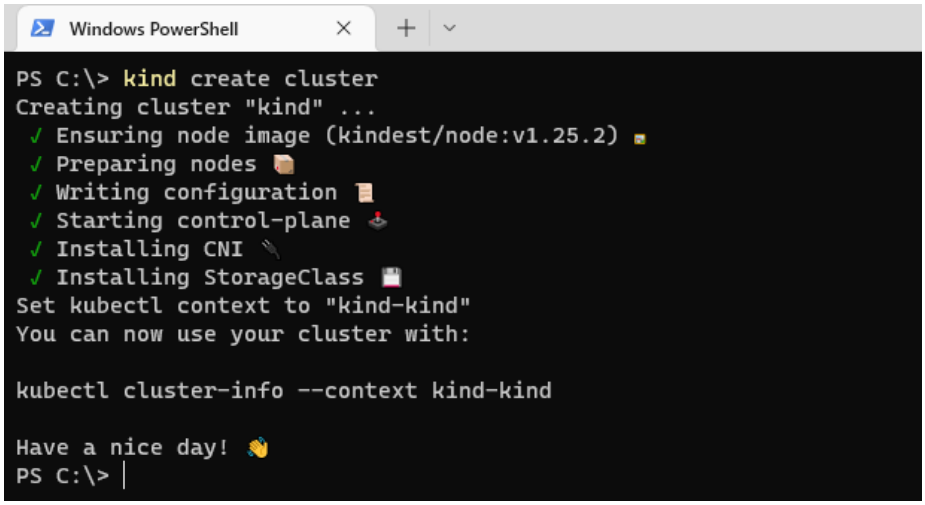

Figura 3: Criando um cluster usando Kind

Este cluster está sendo executado em um container no Docker. Como vemos na figura 4, é possível lista-lo através do comando "docker ps". Como vemos, a porta 6443 está exposta, o kubectl de nossa maquina se conectará na API do kubernetes deste cluster através dela.

[26] https://github.com/kubernetes-sigs/kind/releases

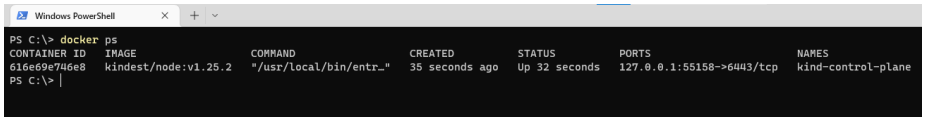

Figura 4: Deletando um cluster usando Kind

Por fim, é possível deletar o cluster criado, apagando toda e qualquer modificação após o final de cada atividade. O comando utilizado é o "kind delete cluster", podemos ver sua execução na figura 5.

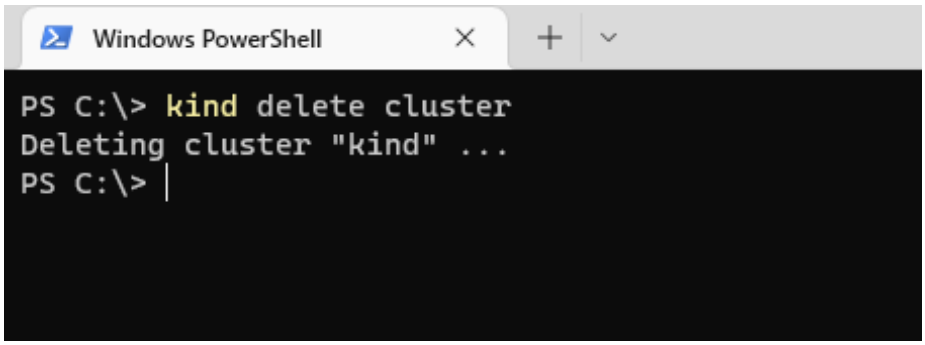

Figura 5: Deletando um cluster usando Kind

3.7 Problemas Conhecidos

No capítulo 9, o StorageClass padrão pode ser diferente em clusters Kubernetes não criados com o Kind. É possível alterar os arquivos, escolhendo o StorageClas correto.

No capítulo 8, a instalação do Ingress depende que a porta 80 e 443 esteja disponível nos servidores que hospedam o cluster Kubernetes. É possível utilizar outras formas de instalação do Ingress Controller para resolver esta questão.

3.8 Conclusão

Os softwares de apoio usados neste documento podem ser alterados conforme a decisão de quem estiver utilizando. Caso isto ocorra, pequenas modificações em comandos podem ser necessárias. É importante notar que o essencial é o acesso administrativo a um cluster Kubernetes através do executável kubectl.

4 Namespaces

Namespaces são divisões lógicas dentro de um cluster Kubernetes. Eles são de extrema importância visto que a maioria dos recursos do Kubernetes devem existir dentro de um Namespace. Namespaces podem ser criados para isolar recursos entre times, podemos também usar Namespaces para separar o ambiente de produção do de homologação sem ter que criar um cluster novo para isto.

Recursos criados dentro de um Namespace devem ter nomes únicos. Porém, podemos repetir nomes de recursos, desde que eles existam em Namespaces diferentes. Podemos considerar namespaces como ambientes completamente isolados. Podemos, através de políticas, definir que recursos em um namespace não tenham acesso a nada de outros namespaces.

Através do comando "kubectl get ns" podemos listar todos os Namespaces do nosso cluster. Aqui devemos dar especial importância para os seguintes namespaces observados na figura 6: O Namespace default, que é o namespace padrão quando não apontamos nossos comandos para nenhum. O Namespace kube-system abarca os recursos de sistema do Kubernetes e o kube-public, onde seus recursos podem ser observados por recursos em qualquer outro Namespace.

```
PS C:\git\apostila\namespaces> kubectl get ns
NAME                 STATUS   AGE
default              Active   80s
kube-node-lease      Active   82s
kube-public          Active   82s
kube-system          Active   82s
local-path-storage   Active   66s
```

Figura 6: Lista de namespaces através do comando kubectl get ns.

Como podemos ver na figura 7, podemos listar todos os recursos existentes no namespace kube-system através do comando "kubectl get all --namespace kube-system". A opção "--namespace", ou "-n" pode ser usada para escolher qual namespace enviaremos o comando.

Esses são os recursos de administração do Kubernetes, e por isso ficam isolados em um namespace só deles. Em geral não devemos criar recursos no namespace kube-system.

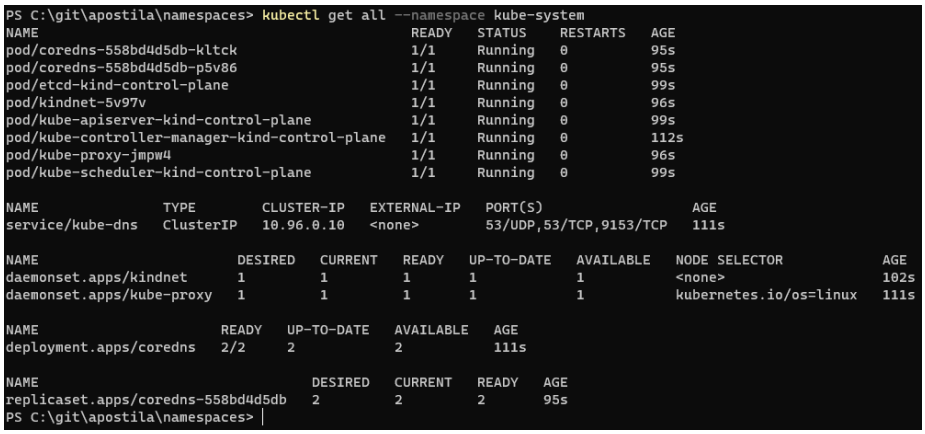

Figura 7: Lista todos os recursos do namespaces kube-system.

4.1 Criando Namespaces

Podemos criar um Namespace através do comando "kubectl create ns" seguido do nome do namespace ou de um arquivo YAML com a descrição do recurso. Na imagem 8 podemos ver um namespace sendo criado com o arquivo namespace/namespace.yaml[27].

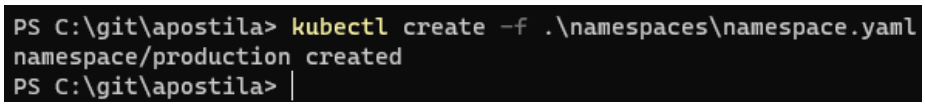

Figura 8: Lista todos os recursos do namespaces kube-system.

O conteúdo do arquivo pode ser visto no código 4.1 e é bem simples, sendo basicamente a definição de um namespace chamado production.

Código 2: O arquivo YAML de um namespace.

```
1 # namespace.yaml
2 apiVersion: v1          (1) Versao da API
3 kind: Namespace         (2) Tipo do recurso
4 metadata:
5   name: production      (3) Nome do recurso
```

A partir disto, podemos criar outros recursos neste namespace através da

[27] Disponível em: github

adição da opção "-n production" ou adicionando o metadata namespace: production nos YAMLs de outros recursos.

Nem todos os recursos do Kubernetes são vinculados a namespaces. Geralmente recursos que são utilizados pelo cluster todo não são vinculados a Namespaces. Um exemplo disto são as classes de Storage, usadas nos recursos vinculados à persistência de dados e os próprios servidores do cluster, chamados de Nodes.

É possível listar todos os recursos não vinculados a namespace através do comando "kubectl api-resources --namespaced=false" e os vinculados a namespaces através do comando "kubectl api-resources --namespaced=true".

4.2 Conclusão

Namespaces são usados para dividir o cluster Kubernetes de diversas formas. Podemos usar Namespaces para dividir o cluster entre times de desenvolvimento, entre aplicações ou entre ambientes de Produção e Homologação.

Em casos específicos, ambientes de produção e desenvolvimento podem ser literalmente clusters diferentes. A questão é, essa divisão pode ser feita em um único cluster através de Namespaces, regras de isolamento e de acesso.

Como em boa parte do Kubernetes, com ele é possível ter a possibilidade de adaptar seu cluster às necessidades de sua equipe ou empresa.

5 POD

Pods são um grupo de um ou mais containers, que compartilham recursos de rede e storage formando um único processo do ponto de vista do Kubernetes e são sempre executados em um mesmo servidor. Um Pod é o menor recurso que podemos criar no Kubernetes.

Conforme Victor Farcic discorre em The DevOps Toolkit 2.3: "Pods são o equivalente aos tijolos que usamos para construir casas. Ambos são monótonos e não fazem muito por eles mesmos. No entanto, eles são os blocos de construção fundamentais sem os quais não poderíamos construir a solução que nos propomos para construir."

Podemos criar um pod de forma manual usando comandos de linha de comando ou descrevê-lo usando YAML. Conforme vemos no Código 5, o YAML que descreve um Pod pode ser bem simples.

No código 5, descrevemos um Pod com um container baseado na imagem nginx:latest[28] com uma porta definida na TCP 80. Definimos um rótulo (ou label). demos um nome e definimos em que Namespace esse Pod irá existir.

Um Pod é um recurso "Namespaced", isto é, ele deve sempre estar relacionado a um namespace e seu nome deve ser único em cada namespace.

Código 3: O YAML de um Pod.

```
1  # pod.yaml
2  apiVersion: v1                (1) Versao da API
3  kind: Pod                     (2) Tipo do recurso
4  metadata:
5    name: static-web            (3) Nome do recurso
6    namespace: default          (4) Namespace onde o recurso vai existir
7    labels:
8      role: myrole              (4) Labels aplicadas ao recurso
9  spec:
10   containers:
11     - name: web               (5) Nome do Container
12       image: nginx:latest     (6) Imagem do Container
13       ports:
14         - name: web           (7) Nome definido para a porta
15           containerPort: 80   (8) Porta TCP definida
```

Pods em geral tem apenas um container. Porém podemos usar Pods com múltiplos containers para casos específicos. Também é possível definir containers de inicialização que fazem desempenham funções específicas durante a

[28] https://hub.docker.com/_/nginx

inicialização do pod e então param ou mesmo container efêmeros usados apenas durante debugs e troubleshoots.

Um ponto muito importante a ser observado é que Containers não são geridos diretamente pelo Kubernetes. O Kubernetes, através do Kubelet, cria Pods que este sim, criam containers. Muitas vezes podemos acabar tratando Pods como Containers, porém eles não são a mesma coisa.

5.1 Criando um pod

Podemos criar um Pod através do comando "kubectl create pod" seguido do nome do pod ou de um arquivo YAML com a descrição do recurso. Na imagem 9 podemos ver um pod sendo criado com o arquivo pod/pod.yaml[29].

```
PS C:\git\apostila> kubectl create -f .\pods\pod.yaml
pod/static-web created
PS C:\git\apostila>
```

Figura 9: O Pod Static-web foi criado com sucesso.

Uma vez criado, o Kubernetes tentará garantir que este Pod estará sempre em execução. Com o comando "kubectl get pods -n default" podemos ver os pods em execução no Namespace default:

```
PS C:\git\apostila> kubectl get pods -n default
NAME         READY   STATUS    RESTARTS   AGE
static-web   1/1     Running   0          24m
PS C:\git\apostila>
```

Figura 10: O Pod static-web tem um container de um pronto, está em execução e sem nenhuma reinicialização ocorrida. Ele foi criado há 24 minutos.

Podemos ver os logs gerados por este pod com o comando "kubectl logs static-web -n default":

Sempre quando usamos comandos ligados a Pods devemos definir em que Namespace a consulta vai ser executada. Caso nenhum namespace seja definido, o comando será executado no Namespace default.

[29] Disponível em: github

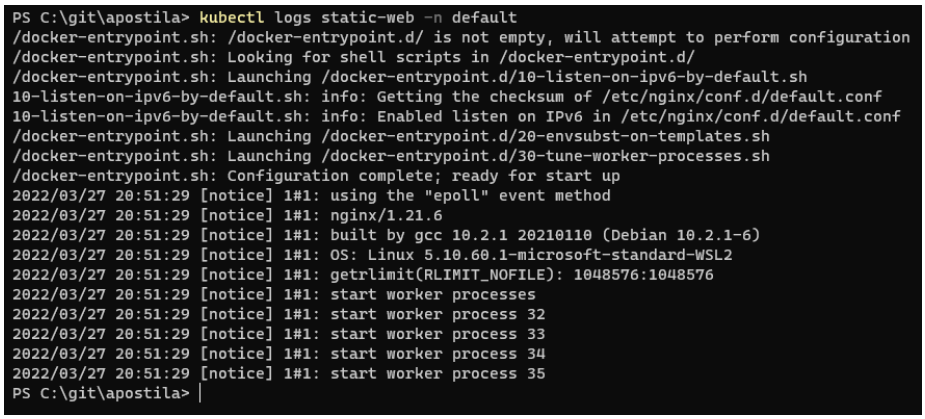

Figura 11: Os logs do Pod static-web.

Como meio de troubleshooting é possível executar comandos ou acessar via linha de comando de forma interativa um Pod. Para isto, podemos usar o comando "kubectl exec".

```
PS C:\git\apostila\pods> kubectl exec -it pods/static-web -- bash
root@static-web:/# hostname
static-web
root@static-web:/#
```

Figura 12: Acessando um pod com o kubectl exec.

Como podemos ver na imagem 12, usamos o comando "kubectl exec -it pods/static-web -- bash" para acessar de forma interativa através do shell Bash o Pod static-web. Nesse comando usamos algumas funções interessantes.

Usamos as opções -i e -t de uma só vez (-it) para dizer que nosso Stdin (nosso teclado) deverá ser direcionado para o Pod (-i) e que está será uma sessão do tipo TTY (-t) com entrada e saída de texto. Seguimos com o sufixo pod/, informando que estamos acessando um pod, seguido de seu nome. Após isto usamos o "--" para separar o comando enviado para o Pod das opções do kubectl exec. Para saber outras opções disponíveis podemos usar o comando "kubectl exec -h".

De forma simples, poderíamos pensar que para escalar uma aplicação devemos criar Pods com múltiplos containers. Porém esta não é a forma correta de se fazer isto. O método correto é na verdade criar vários pods! Para isto,

usaremos Deployments[30].

5.2 Conclusão

Neste capítulo falamos sobre Pods, os blocos de criação de um Cluster Kubernetes. Eles são muito úteis e são extremamente customizáveis. Através de seu uso nós orquestramos aplicações através do Kubernetes.

Porém como dito, os Pods são o início, para poder realmente utilizar as vantagens da orquestração de containers, precisaremos de recursos mais avançados. Nos próximos capítulos os recursos responsáveis por dar mais complexidade e utilidade aos Pods serão explorados.

[30] Deployment: Um recurso do Kubernetes.

6 Deployment

Pods não são feitos para serem iniciados diretamente e tão pouco devem ser usados para escalar Containers em diversas réplicas. Pods são como objetos isolados com um ou mais containers em seu interior. Cada container presta um serviço, como por exemplo: Um container executando um serviço web e outro executando um sistema de captura e envio de logs deste mesmo serviço web.

Ambos os containers devem ser executados em conjunto. Essa é a função de um Pod, garantir a execução de ambos os containers de forma orquestrada. Agora, escalar esse serviço em várias réplicas é papel de outros recursos.

Antes de adentrar no recurso Deployment devemos falar do ReplicaSet. O papel de um ReplicaSet é garantir que um número determinado de réplicas de um Pod esteja sempre em execução. Com isto, caso uma das réplicas seja parada inesperadamente, o ReplicaSet irá iniciar um novo Pod até que o número de réplicas definido seja alcançado.

Continuando, ReplicaSets são responsáveis por escalar e garantir a saúde de serviços que usam Pods. Já Deployments são responsáveis por gerir ReplicaSets. É possível criar ReplicaSets diretamente, porém usando Deployments nós conseguimos usar recursos mais avançados, principalmente quando precisamos aplicar atualizações em nossos componentes.

Podemos ver no Código 6 o arquivo YAML que descreve um Deployment que criará um ReplicaSet que manterá 3 pods com containers do Nginx. É muito importante notar alguns pontos. Podemos ver que ao criar o Deployment, definimos em (7) um Seletor. Com isto, todos os Pods do Namescape Production (4) que tiveram os Labels (Rótulos) app = nginx serão gerenciados pelos ReplicaSets criados por este Deployment. Com isto, ao descrever os Pods deste Deployment, em (10) definimos através do Template que todos os pods devem ter os Labels app = nginx (9).

Código 4: O arquivo YAML de um Deployment

```
1  # deployment.yaml
2  apiVersion: apps/v1                    (1) Versao da API
3  kind: Deployment                       (2) Tipo do recurso
4  metadata:
5    name: nginx--deployment              (3) Nome do recurso
6    namespace: production                (4) Namespace do recurso
7    labels:
8      app: nginx                         (5) Rotulos aplicadas ao recurso
9  spec:
10   replicas: 3                          (6) Numero de replicas
11   selector:                            (7) Seletor usado pelo ReplicaSet
12     matchLabels:                       (8) Tipo de seletor
```

```
13        app: nginx                    (9) Rotulos usados no seletor
14    template:                         (10) Template usado na criacao dos pods
15      metadata:
16        labels:
17          app: nginx                  (11) Rotulos que todos os pods devem ter
18      spec:
19        containers:                   (12) Containers do Pod
20        - name: nginx                 (13) Nome do Container
21          image: nginx:1.14.2         (14) Imagem do Container
22          ports:
23          - containerPort: 80         (15) Porta usada pelo Container
```

É muito interessante verificar como a descrição de pods através de arquivos YAML se parecem com a descrição dos containers em um Deployment. Isso se dá porque o Kubernetes reutiliza seus recursos, logo o recurso Deployment cria o recurso ReplicaSet que cria o recurso Pod. Assim, ao descrever um deles, os outros acabam sendo referenciados.

6.1 Criando Deployments

Podemos criar um Deployments através de um arquivo YAML com a descrição do recurso. Na imagem 9 podemos ver um pod sendo criado com o arquivo deployment/deployment.yaml[31]. Também é possível criar Deployments diretamente pela linha de comando, porém devido a sua complexidade, em geral Deployments são criados através de arquivos YAML.

```
PS C:\git\apostila> kubectl create -f .\deployments\deployment.yaml -n production --save-config
deployment.apps/nginx-deployment created
```

Figura 13: Um Deployment sendo criado com o comando kubectl create.

Podemos ver na figura 13 um Deployment sendo criado através de um arquivo de configuração. A opção "--save-config" foi adicionada para que as configurações definidas no arquivo sejam salvas no objeto, assim poderemos aplicar atualizações neste deployment.

Após criar o Deployment nginx-deployment, podemos listar todos os recursos criados no namespace production. Com isto poderemos verificar tudo que foi criado ao criarmos um deployment. Na figura 14 podemos ver o ReplicaSet e Pods criados.

A principal vantagem de um Deployment para um ReplicaSet é a capacidade de ser atualizado. Com o comando "kubectl apply" podemos aplicar uma atu-

[31] Disponível em: github

```
PS C:\git\apostila> kubectl get all -n production
NAME                                    READY   STATUS    RESTARTS   AGE
pod/nginx-deployment-66b6c48dd5-7cxd6   1/1     Running   0          24s
pod/nginx-deployment-66b6c48dd5-9wctx   1/1     Running   0          24s
pod/nginx-deployment-66b6c48dd5-grhgz   1/1     Running   0          24s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment    3/3     3            3           24s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-deployment-66b6c48dd5    3         3         3       24s
```

Figura 14: Todos os recursos do namespace production

alização diretamente em um Deployment. Na figura 15 nós aplicamos o arquivo "deployment_update.yaml" que altera a imagem do Pod nginx. Esse tipo de atualização é extremamente comum em ambientes que usam o Kubernetes e o Deployment ajuda a orquestrar esse tipo de operação.

```
PS C:\git\apostila> kubectl apply -f .\deployments\deployment_update.yaml -n production
deployment.apps/nginx-deployment configured
```

Figura 15: Aplicando uma atualização em um Deployment

Após aplicar esta atualização nós podemos verificar novamente os recursos criados no namespace production. Na figura 16 podemos observar que um novo ReplicaSet e 3 novos pods estão em execução em nosso ambiente. Isso se dá pois o Deployment criou um novo ReplicaSet com a nova imagem definida e manteve o anterior para caso um rollback[32] seja necessário.

```
PS C:\git\apostila> kubectl get all -n production
NAME                                    READY   STATUS    RESTARTS   AGE
pod/nginx-deployment-75b69bd684-57xtd   1/1     Running   0          2m7s
pod/nginx-deployment-75b69bd684-725qt   1/1     Running   0          2m19s
pod/nginx-deployment-75b69bd684-p7kv4   1/1     Running   0          2m8s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment    3/3     3            3           3m17s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-deployment-66b6c48dd5    0         0         0       3m17s
replicaset.apps/nginx-deployment-75b69bd684    3         3         3       2m20s
```

Figura 16: Todos os recursos do namespace production após a atualização

Um ponto muito importante que devemos observar nos Deployments é a

[32]Rollback: Voltar atrás em uma atualização para a versão anterior.

estratégia de atualização, ela é definida pelo campo StrategyType e por padrão é definida como RollingUpdate, também chamado de zero downtime rollouts[33].

Na estratégia RollingUpdates, um novo ReplicaSet é criado com 0 réplicas. Após isto, uma réplica é criada. Após este Pod ser iniciado corretamente, uma réplica do ReplicaSet anterior é deletado. Esse procedimento é repetido até o novo ReplicaSet ter todas as réplicas ativas e o antigo todas desligadas. Em casos de deployments com muitas réplicas, esse procedimento por padrão só pode acontecer com no máximo 25% das réplicas ao mesmo tempo.

Também é possível definir a estratégia como Recreate, onde as réplicas do ReplicaSet antigo são destruídas antes de serem recriadas no novo ReplicaSet. Neste modelo o Deployment ficará inoperante durante a troca de réplicas.

Na figura 17 podemos observar a descrição do Deployment nginx-deployment após a atualização. Isso foi feito através do comando "kubectl describe" que pode ser usado em qualquer recurso do Kubernetes. Nessa imagem podemos ver a estratégia definida e nos eventos, os ReplicaSets sendo modificados.

```
PS C:\git\apostila> kubectl describe deploy nginx-deployment -n production
Name:                   nginx-deployment
Namespace:              production
CreationTimestamp:      Sat, 09 Apr 2022 16:46:41 -0400
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 2
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:latest
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-75b69bd684 (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  76s   deployment-controller  Scaled up replica set nginx-deployment-66b6c48dd5 to 3
  Normal  ScalingReplicaSet  19s   deployment-controller  Scaled up replica set nginx-deployment-75b69bd684 to 1
  Normal  ScalingReplicaSet  8s    deployment-controller  Scaled down replica set nginx-deployment-66b6c48dd5 to 2
  Normal  ScalingReplicaSet  8s    deployment-controller  Scaled up replica set nginx-deployment-75b69bd684 to 2
  Normal  ScalingReplicaSet  7s    deployment-controller  Scaled down replica set nginx-deployment-66b6c48dd5 to 1
  Normal  ScalingReplicaSet  7s    deployment-controller  Scaled up replica set nginx-deployment-75b69bd684 to 3
  Normal  ScalingReplicaSet  6s    deployment-controller  Scaled down replica set nginx-deployment-66b6c48dd5 to 0
```

Figura 17: Todos os recursos do namespace production após a atualização

[33] Algo como atualização sem tempo de inoperância.

6.2 Desfazendo atualizações com Deployment Rollbacks

Nem sempre uma atualização acontece da maneira que gostaríamos. Por isso, poder voltar atrás em um update é extremamente importante. Essa também é uma das funções do Deployment. Através do comando "kubectl rollout history" é possível listar as "revisões" aplicadas a um Deployment. Isso pode ser visto na figura 18.

```
PS C:\git\apostila> kubectl rollout history deployment/nginx-deployment -n production
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

Figura 18: Histórico de atualização do Deployment nginx-deployment

A partir das revisões vistas na figura 18 podemos aplicar um Rollback[34] definindo qual revisão decidimos que deve ser aplicada. Para isso podemos usar o comando " kubectl rollout undo" com a opção --to-revision, escolhendo a revisão que queremos. Na figura 19 podemos observar o procedimento.

```
PS C:\git\apostila> kubectl rollout undo deployment/nginx-deployment --to-revision=1 -n production
deployment.apps/nginx-deployment rolled back
```

Figura 19: Desfazendo uma atualização em um Deployment

Ao final, na figura 20 podemos observar que o ReplicaSet antigo está novamente com 3 réplicas. Com isso, a troca da imagem do nginx foi desfeita.

```
PS C:\git\apostila> kubectl get all -n production
NAME                                    READY   STATUS    RESTARTS   AGE
pod/nginx-deployment-66b6c48dd5-mmpjk   1/1     Running   0          20s
pod/nginx-deployment-66b6c48dd5-p2l7w   1/1     Running   0          23s
pod/nginx-deployment-66b6c48dd5-t6h2x   1/1     Running   0          21s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment   3/3     3            3           44m

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-deployment-66b6c48dd5   3         3         3       44m
replicaset.apps/nginx-deployment-75b69bd684   0         0         0       43m
```

Figura 20: Recursos após aplicação de um Rollback.

[34] Voltar ou desfazer algo que foi feito.

6.3 Conclusão

Neste capítulo nós abordamos os Deployments, recursos responsáveis por orquestrar aplicações com capacidade de serem atualizadas de maneira programática. Vimos que Deployments controlam ReplicaSets que por sua vez controlam Pods.

Por fim, vimos como uma atualização afeta os ReplicaSets geridos por um Deployment e como desfazer essas atualizações. Com isso, abordamos vários aspectos do gerenciamento de uma aplicação no Kubernetes. Nesse momento, temos uma aplicação web com 3 réplicas iniciadas, com capacidade de atualização sem downtime pronta e com capacidades de self-healing.

No próximo capítulo descobriremos como acessar nossa aplicação.

7 Service

Service é o recurso do Kubernetes responsável por garantir uma comunicação estável e confiável entre os Pods de nosso Cluster. Em um cluster Kubernetes cada Pod recebe um endereço IP que possibilita a comunicação de rede entre eles. Além disso, containers que compartilham um Pod podem se comunicar internamente através do uso do localhost.

Levando em conta que Pods são extremamente voláteis e que adicionando a camada dos Deployments que vimos no capítulo anterior, mesmo atualizações irão causar a substituição dos Pods, gerando então mudanças nos IP's da aplicação. Isso tornaria muito complexa a gerência de IP's dentro de nosso ambiente.

Para isto, o Kubernetes implementou os Services. Com eles, definimos um recurso que cria um IP fixo para nossos Pods, ou conjunto de Pods. Com ele, conseguimos englobar todas as réplicas de um ReplicaSet gerenciado por um Deployment em um único IP fixo que pode ser usado com tranquilidade nas comunicações internas ou externas do Cluster.

Além disto, através do uso de um Addon[35], o CoreDNS, os Services também ganham um nome DNS válido dentro do Cluster, o que facilita em muito a descoberta de serviços e comunicação entre os usuários da aplicação e os Pods.

O Kubernetes usa o kube-controller-manager na figura do controlador Endpoints Controller para popularizar os Services com Endpoints, ou seja, o endereço IP dos Pods que pertencem a este Service. Para isto, da mesma forma que os Deployments, usamos Selectors para definir Labels (Rótulos) que distinguiram os Pods a serem escolhidos. É muito comum usar os mesmos Selectors usados no Deployment para facilitar o gerenciamento do ambiente.

Existem atualmente 5 tipos de Services, a saber: ClusterIP, NodePort, ,LoadBalancer, ExternalService, Headless. Cada um deles tem um uso específico dentro do Cluster e em certos casos, podem até ser utilizados em conjunto para o mesmos Pods.

7.1 ClusterIP

Um Service do tipo ClusterIP é criado quando o acesso a este serviço só será feito dentro do Cluster. Para isto, o Kubernetes irá criar um endereço IP e DNS válidos apenas internamente dentro do cluster. Com isto, será possível acessar os serviços dentro do cluster através de um nome DNS fácil de usar e lembrar.

[35] Recursos que adiciona nova funções ao cluster

Código 5: O YAML de um service do tipo ClusterIP

```
1  # service-clusterip.yaml
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: nginx-clusterip-service
6    namespace: production
7  spec:
8    selector:
9      app: nginx
10   type: NodePort
11   ports:
12     - name: http
13       port: 80
14       targetPort: 80
15       nodePort: 30080
```

Este tipo de Service é muito útil por exemplo para prover a comunicação entre o banco de dados e uma aplicação. Como em geral o banco de dados de uma aplicação é acessado apenas por esta aplicação, um ClusterIP proverá esse acesso sem expor o banco para fora do cluster Kubernetes.

7.2 NodePort

Um Service do tipo NodePort é criado quando o acesso a este serviço deverá ser feito tanto internamente quanto externamente ao Cluster. Para fazer este papel, devemos definir uma porta TCP que ficará disponível em todos os servidores do Cluster. Qualquer acesso a essa porta será enviado para este Service.

Isso é possível pois todos os servidores do cluster tem o kube-proxy rodando, criando então uma rede virtual que serve de proxy entre os servidores e os Services do tipo NodePort. Por padrão o Kubernetes separa range de portas TCP 30000-32767, porém podemos definir manualmente a porta através da opção nodePort.

Código 6: O YAML de um service do tipo NodePort

```
1  # service-clusterip.yaml
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: nginx-nodeport-service
6    namespace: production
7  spec:
8    selector:
```

```
9       app: nginx
10    type: NodePort
11    ports:
12      - name: http
13        port: 80
14        targetPort: 80
15        nodePort: 30080
```

Esse tipo de Service pode ser usado para prover acesso externo a aplicações. Porém para isto, devemos expor os endereços dos servidores do Cluster para acesso externo ou usar um balanceador de carga em frente aos servidores do cluster Kubernetes.

7.3 LoadBalancer

Os Services do tipo LoadBalancer, assim como os do tipo NodePort servem para prover acesso externo a nossos serviços. Porém diferente dos NodePorts, os LoadBalancers usam integrações com Provedores de Computação em Nuvem ou Addons em nossos cluster para adicionar um IP externo válido diretamente no nosso Service. Com isto, nosso serviço passará a ter um ClusterIP válido internamente no cluster e um ExternalIP válido externamente no Cluster.

No caso das versões do Kubernetes gerenciadas por provedores de Computação em Nuvem, ao criar um Service do tipo LoadBalancer, o provedor irá providenciar um IP válido para a internet que poderá ser acessado de qualquer outro computador conectado à Internet. Existem várias distribuições do Kubernetes gerenciados pela nuvem, os principais são o GKS da Google, AKS do Azure e EKS da AWS.

Código 7: O YAML de um service do tipo LoadBalancer

```
1  # service-loadbalancer.yaml
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: nginx-service
6    namespace: production
7  spec:
8     selector :
9       app: nginx
10   type: LoadBalancer
11   ports:
12     - name: http
13       port: 80
14       targetPort: 80
```

7.4 ExternalService e Headless

Os Services do tipo ExternalService e Headless são em geral menos usados e servem para pontos muito específicos. Os ExternalServices servem para referenciar endereços de fora do Cluster Kubernetes como Services do Kubernetes. Isto pode ser útil para customizar serviços criados dentro do Kubernetes com recursos legados.

Código 8: O YAML de um service do tipo ExternalName

```
1  # service-externalservice.yaml
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: external-db-service
6    namespace: production
7  spec:
8    type: ExternalName
9    externalName: mysql-instance.randomsid.us-east-1.rds.amazonaws.com
```

Já Services do tipo Headless são usados quando não necessitamos que o Service balanceie as conexões. Eles podem ser uteis em casos onde queremos que o endereço DNS do service aponte diretamente para o Pod a ele vinculado, sem balanceamento ou proxy.

Para criar um Service do tipo headless devemos apenas definir como "None" a opção "clusterIP"[36]

Código 9: O YAML de um service do tipo Headless

```
1  # service-headless.yaml
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: nginx-headless-service
6    namespace: production
7  spec:
8    clusterIP: None
9    selector:
10     app: nginx
11   type: ClusterIP
12   ports:
13     - name: http
14       port: 80
15       targetPort: 80
```

[36] esse campo é em geral populado automaticamente pelo Kubernetes.

Esse tipo de Service é muito usado em aplicações que devem manter estado em disco.

7.5 Criando Services

Services são recursos muito importantes do Kubernetes. Aplicações sem Services dificilmente podem ser acessadas, tornando sua utilidade praticamente nula.

Para criar um Service, primeiro devemos nos atentar a quais Pods esse Service proverá acesso. Assim como os Deployments, usaremos Selectors baseados em Labels para escolher os Pods. Com isto, iniciaremos criando um Deployment. No código 7.5 vemos o Deployment da aplicação WhoAmI, que será muito útil neste caso. Ela é uma aplicação Web que como resposta para uma solicitação retorna informações úteis sobre o ambiente onde ela está hospedada.

Código 10: O YAML de um service do tipo LoadBalancer

```
1  # deployment_whoami.yaml
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: whoami-deployment
6    namespace: production
7    labels:
8      app: whoami
9  spec:
10   replicas: 2
11   selector:                    (1) Selectors para este Deployment
12     matchLabels:               (2) Selector baseado em Labels
13       app: whoami              (3) Labels definidas
14   template:                    (4) Template para criacao dos Pods
15     metadata:
16       labels:
17         app: whoami            (5) Labels definidos
18     spec:
19       containers:
20       - name: whoami
21         image: traefik/whoami:v1.8.0
22         ports:
23         - containerPort: 80
```

Para criar o Deployment, usaremos o comando "kubectl create -f deployments/deployment_whoami.yaml -n production --save-config", conforme podemos ver na imagem 21:

Conforme vemos na imagem 22 podemos verificar os Pods criados usando o

Figura 21: Um Deployment sendo criado com o comando kubectl create.

comando "kubectl get pods -n production". Os nomes destes Pods são importantes para testarmos as capacidades de balanceamento de carga dos Services.

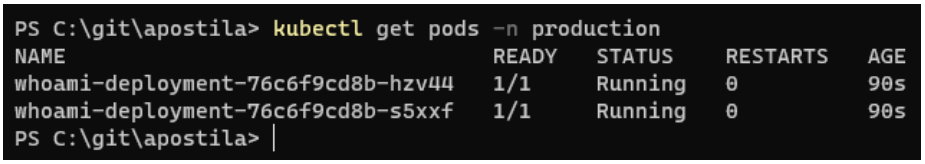

Figura 22: Os Pods criados para o Deployment whoami.

Seguindo, devemos criar um Service do tipo ClusterIP para prover acesso balanceado a esses Pods apenas dentro do cluster Kubernetes. Para isto, usaremos o comando "kubectl create -f services/service-whoami.yaml -n production --save-config".

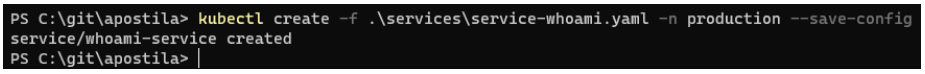

Figura 23: Criando o Service whoami-service.

É importante notar no 7.5 que os seletores (1) devem referenciar os usados no template do 7.5. É assim que o Endpoits Controller irá saber o IP de quais Pods popular no Service criado.

Código 11: O YAML de um service do tipo ClusterIP

```
1  # service_whoami.yaml
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: whoami-service
6    namespace: production
7  spec:
8    selector:
9      app: whoami                (1) Labels definidos para os seletores
10   type: ClusterIP              (2) Tipo do Service
11   ports:
12     - name: http
```

```
13      port: 80                (3) Porta TCP no Service
14      targetPort: 80          (4) Porta TCP no Pod
```

Após criá-lo, podemos descrevê-lo usando o comando "kubectl describe service/whoami-service -n production". Na 24 podemos observar alguns campos importantes. Em IP temos o IP virtual definido para este serviço. No campo Endpoints, podemos ver os endereços IP dos Pods selecionados pelo Endpoints Selector.

```
PS C:\git\apostila> kubectl describe service/whoami-service -n production
Name:              whoami-service
Namespace:         production
Labels:            <none>
Annotations:       <none>
Selector:          app=whoami
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.96.241.98
IPs:               10.96.241.98
Port:              http  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.0.8:80,10.244.0.9:80
Session Affinity:  None
Events:            <none>
PS C:\git\apostila>
```

Figura 24: Criando o Service whoami-service.

Para testar o Service que acabamos de criar, temos algumas opções, a primeira delas é acessar um Pod de nosso ambiente através do comando "kubectl exec". A imagem traefik/whoami:v1.8.0 é muito enxuta, por isso ela não tem um shell em que possamos conectar para testar o Service. Para isso então, criaremos um novo Pod com um shell.

Usaremos para isso o comando "kubectl create -f pods/pod.yaml -n production --save-config", que criará um Pod com um container do Nginx que servirá a nossa proposta.

```
PS C:\git\apostila> kubectl get pods -n production
NAME                                READY   STATUS    RESTARTS   AGE
whoami-deployment-76c6f9cd8b-hzv44   1/1     Running   0          90s
whoami-deployment-76c6f9cd8b-s5xxf   1/1     Running   0          90s
PS C:\git\apostila>
```

Figura 25: Criando um Pod para teste do Service.

Agora, com o comando "kubectl exec -it pod/static-web -n production --

bash" acessaremos o Pod criado e executaremos um acesso a porta 80 do IP do Service usando o curl. Na 26 podemos ver que após fazer alguns acessos ao endereço http://10.96.241.98[37] temos como resposta algumas informações do Pod que respondeu à requisição.

```
PS C:\git\apostila> kubectl exec -it pod/static-web -n production -- bash
root@static-web:/# curl http://10.96.241.98
Hostname: whoami-deployment-76c6f9cd8b-s5xxf
IP: 127.0.0.1
IP: ::1
IP: 10.244.0.9
IP: fe80::2415:9aff:fed9:3671
RemoteAddr: 10.244.0.10:49178
GET / HTTP/1.1
Host: 10.96.241.98
User-Agent: curl/7.74.0
Accept: */*

root@static-web:/# curl http://10.96.241.98
Hostname: whoami-deployment-76c6f9cd8b-hzv44
IP: 127.0.0.1
IP: ::1
IP: 10.244.0.8
IP: fe80::80f0:dcff:fe64:3191
RemoteAddr: 10.244.0.10:49282
GET / HTTP/1.1
Host: 10.96.241.98
User-Agent: curl/7.74.0
Accept: */*
```

Figura 26: Acessando o Service com o curl.

Em especial, devemos observar como em cada uma das requisições um Pod diferente respondeu à requisição. O hostname, ou seja, o nome do "servidor" que respondeu a requisição é o mesmo do Pod.

Um segundo modo de testarmos o acesso a este Service é o uso do comando "kubectl port-forward". Com este comando, através do kubectl nós criaremos um proxy entre uma porta de nosso computador e a porta de algum recurso no cluster Kubernetes. Este comando é muito útil para testes de aplicações publicadas no nosso cluster.

Conforme vemos na 27, usaremos o comando "kubectl port-forward svc/whoami-service 8080:80 -m production" para criar a conexão entre a porta 8080 de nosso computador com a porta 80 do Service whoami-service do Namespace production:

Após isto, como vemos na 28, podemos usar um navegador em nosso com-

[37] O IP do Service visto no campo IP.

```
PS C:\git\apostila> kubectl port-forward svc/whoami-service 8080:80 -n production
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
Handling connection for 8080
```

Figura 27: Criando um proxy com o comando kubectl port-forward.

putador para acessar o Service, que balanceará as requisições entre nossos Pods.

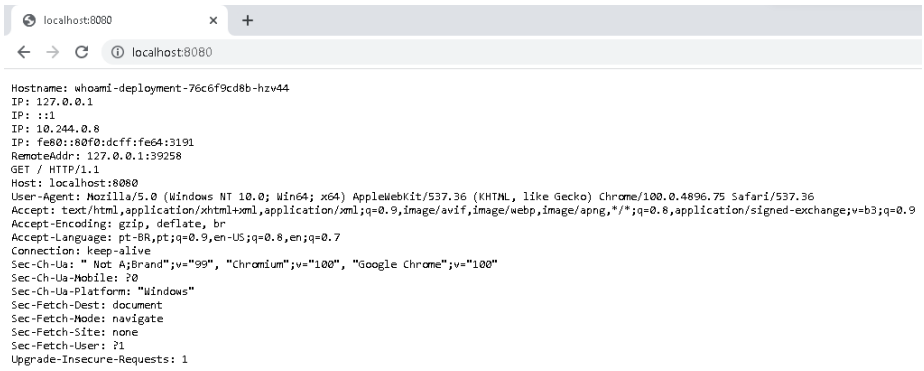

Figura 28: Acessando o Service em nosso navegador.

Ainda sobre os Services, lembrar de um endereço IP para acessar uma aplicação em nosso ambiente não é a melhor forma de se trabalhar. Para resolver isto, todo Service criado no Kubernetes recebe um endereço DNS interno. O DNS interno segue o seguinte padrão ao criar os endereços: ¡nome-do-service¿.¡namespace-do-service¿.svc.cluster.local. Sendo assim, nosso Service poderia ser acessado através do endereço whoami-service.production.svc.cluster.local.

Na 29 podemos ver isso acontecendo:

Além disto, para recursos criados dentro do mesmo Namespace, podemos abreviar o nome DNS para apenas o nome do Service! Isso é possível pois todo Pod em nosso ambiente recebe as configurações do servidor DNS interno do Kubernetes com zonas de pesquisa propriamente configuradas para pesquisar usando os domínios "¡namespace-do-service¿.svc.cluster.local" ".svc.cluster.local" "cluster.local". Conforme vemos na 30, nomes como whoami-service e whoami-service.production também apontaram para o IP correto.

```
root@static-web:/# curl http://whoami-service.production.svc.cluster.local
Hostname: whoami-deployment-76c6f9cd8b-s5xxf
IP: 127.0.0.1
IP: ::1
IP: 10.244.0.9
IP: fe80::2415:9aff:fed9:3671
RemoteAddr: 10.244.0.10:50026
GET / HTTP/1.1
Host: whoami-service.production.svc.cluster.local
User-Agent: curl/7.74.0
Accept: */*
```

Figura 29: Acessando o Service através de um DNS.

```
root@static-web:/# cat /etc/resolv.conf
search production.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
root@static-web:/#
```

Figura 30: Configuração DNS de um Pod do namespace production.

7.6 Conclusão

Neste capítulo abordamos os Services, seus tipos e suas utilidades. Os Services são recursos de extrema importância em nosso Cluster, servindo tanto para facilitar a comunicação entre nossas aplicações ou para dar acesso a essas aplicações ao mundo exterior.

Services podem facilitar a comunicação dentro de nosso Cluster. Seu uso é de extrema importância para aplicações compostas por diversos Deployments.

Além disso, Services podem ser usados para prover acesso a usuários externos do Cluster, tendo a capacidade de até mesmo interagir com balanceadores de carga de provedores de Computação em Nuvem, adicionando maior controle à entrada de dados.

8 Ingress

Dar acesso a aplicações dentro de nossos Clusters Kubernetes é papel dos Services. A combinação entre Services do tipo ClusterIP e LoadBalancer em geral resolvem grande parte das necessidades de acesso via rede das mais complexas aplicações. Porém, conforme o número de aplicações aumenta, o número de endereços IP públicos também aumenta.

Além disso, os Services são recursos de camada 4[38], não atuando diretamente no interior dos pacotes que trafegam pelo kube-proxy. Em certos casos, coisas como customização de headers HTTP ou direcionamento de tráfego baseado em paths não são possíveis com Services.

Para isto, um recurso customizado foi criado. De tão popular, uma nova categoria de recursos foi criada para abarcar esta solução. Os Ingresses são aplicações que atuam como Proxys HTTP de borda entre os acessos externos e os Services internos de nossas aplicações.

São chamados Proxys de Borda pois ficam na borda, ou seja, na fronteira de nosso Cluster, analisando, aplicando modificações e transportando os pacotes HTTP (ou mesmo TCP), que entram em nosso Cluster. Em geral eles observam, através da API do Kubernetes a criação de recursos do tipo Ingress onde definimos a relação entre endereços e paths com Services de nosso Cluster.

Código 12: O YAML de um Ingress.

```
1  #ingress.yaml
2  apiVersion: networking.k8s.io/v1        (1) API do Kubernetes usada
3  kind: Ingress                           (2) Tipo do recurso
4  metadata:
5    name: minimal-ingress                 (3) Nome do recurso
6    annotations:                          (4) Anotacoes que geram modificacoes
7      nginx.ingress.kubernetes.io/rewrite-target: /
8  spec:
9    rules:                                (6) Inicio das regras
10   - http:                               (7) Regras do tipo HTTP
11       paths:                            (8) Regras baseadas em path
12       - path: /testpath                 (9) Path escolhido
13         pathType: Prefix                (10) Tipo de path
14         backend:                        (11) Inicio do bloco de backend
15           service:                      (12) Escolher um Service
16             name: test                  (13) Nome do Service
17             port:
18               number: 80                (14) Porta do Service
```

[38] 4ª camada no modelo OSI, Transporte.

No 8 podemos observar um Ingress simples. Nele definimos que todo acesso que chegue no endereço http://*/testpath* (Qualquer endereço seguido de /testpath seguido de qualquer coisa) seja redirecionado ao Service test na porta 80. Nele aplicamos também a Annotation "nginx.ingress.kubernetes.io/rewrite-target" que reescreve a requisição para que ela chegue como http://*/ no Service.

É possível usar praticamente qualquer mistura entre verbos, endereços e paths existentes no padrão HTTP para direcionar os acessos a nossa aplicação. Isso abre margem para a criação de topologias extremamente complexas que necessitam de muitos arquivos de configuração e serviços de apoio.

Diferente de outros recursos do Kubernetes, que em geral rodam no kube-controller-manager, não existe um Ingress Controller por padrão. O que o Kubernetes nos entrega é um recurso chamado Ingress que através da API do Kubernetes pode ser observado e gerenciado. Com isto, a comunidade criou uma serie de Ingress Controllers de código aberto que podem ser usados em nosso ambiente.

Alguns dos Ingress Controllers mais usados são:

- Nginx Ingress
- Traefik
- Kong
- Ambassador

Fica a cargo do administrador do cluster Kubernetes escolher o que mais atenda às suas necessidades através das várias formas de instalação possíveis.

8.1 Criando um Ingress

Para criar um Ingress, primeiro devemos instalar um Ingress Controller em nosso cluster Kubernetes. Existem vários disponiveis no mercado, porém o mais utilizado hoje é provavelmente o Nginx Ingress, que é mantido pela mesma comunidade que mantém o próprio Kubernetes.

Para instalá-lo usaremos a sua versão customizada para uso nos clusters Kubernetes criados com o Kind. Como vemos na 31, iniciaremos criando um cluster Kind que suporte a um Ingress Controller. O arquivo cluster.yaml tem customizações que adicionam suporte ao Ingress Controller e conectam a porta 80 do cluster Kubernetes com a porta 80 de nosso computador.

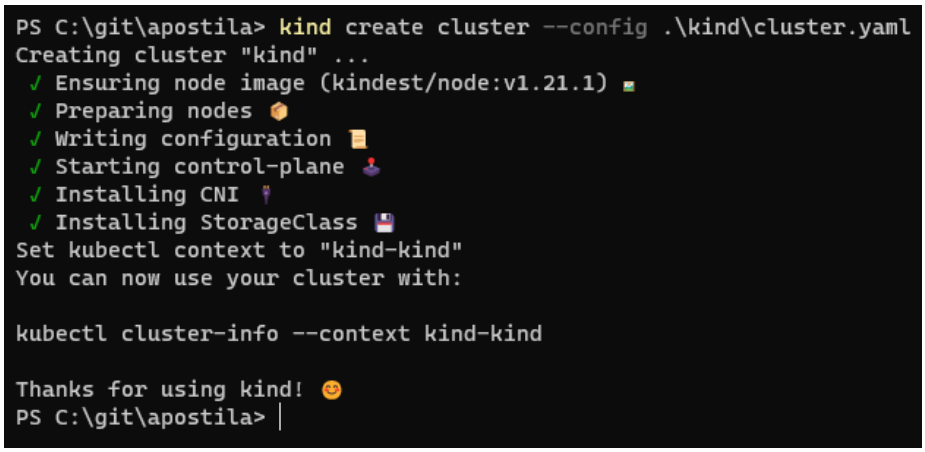

Figura 31: Criando um cluster com o Kind.

Após criar o cluster, devemos criar os recursos que serão acessados através do Ingress. Para isto, usaremos os recursos usados nos capítulos anteriores. Agora criaremos os recursos da aplicação WhoAmI. Podemos ver isso na 32:

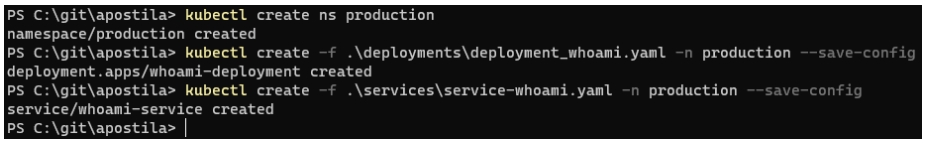

Figura 32: Todos os recursos do namespace production

E agora devemos criar o Ingress. Para isto usaremos o comando "kubectl create -f ./ingress/ingress.yaml -n production --save-config". Isso irá criar um recurso do tipo Ingress, o Ingress Controller verá esse novo recurso e se passará a entregar os acessos ao endereço "http://localhost/testpath" para os Pods do Deployment whoami-deployment através do Service whoami-service.

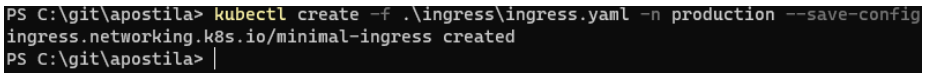

Figura 33: Criando um Ingress.

Podemos então acessar o endereço mencionado para testar se o Ingress foi configurado com êxito:

Por fim, conforme vemos na 35, podemos verificar todos os Ingresses criados

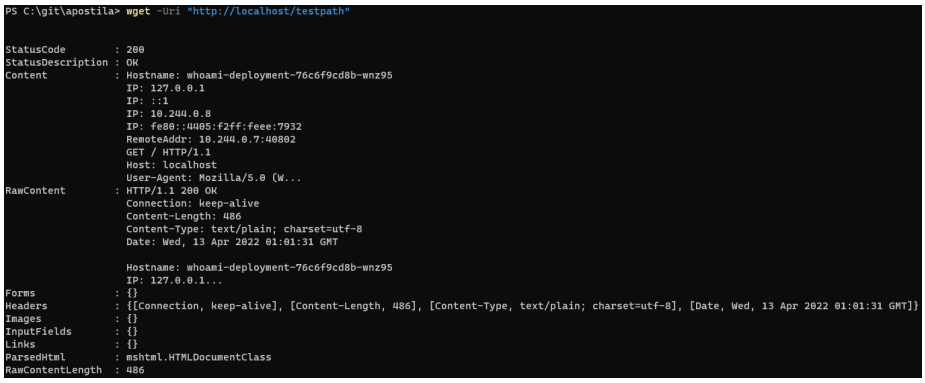

Figura 34: Acessando um Pod através de um Ingress.

para um namespace usando o comando "kubectl get ingress -n production":

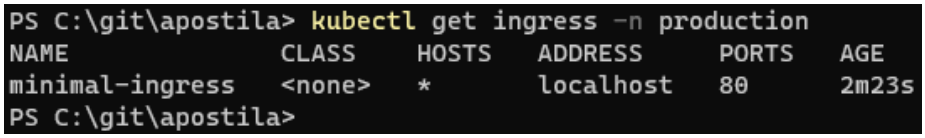

Figura 35: Verificando os Ingress criados para um Namespace.

8.2 Conclusão

Neste capítulo nós vimos o funcionamento dos Ingress Controllers. Seu uso torna mais prático o acesso das aplicações em nosso Cluster através de uma forma unificada e prática. Além disso, os Ingress Controllers adicionam funções mais avançadas a nossos cluster Kubernetes.

Além das funções expostas neste capítulo, os Ingress Controllers oferecem muitas outras. Algumas delas, como a fácil adição de certificados digitais em endereços web ou a manipulação de tráfego HTTP avançado, tornam quase que essenciais a maioria dos cluster Kubernetes. Cabe aos administradores do Cluster escolher a melhor opção em cada situação.

9 Persistência de dados

Do ponto de vista do Kubernetes existem dois tipos de aplicações: As com persistência e as sem persistência em disco. Toda aplicação mantém seu estado em algum lugar. Seja na memória RAM, em um banco de dados relacional, em um bucket S3[39], ou diretamente no disco rígido do servidor que a hospeda.

Quando tratamos de Containers, aplicações que mantém estado em disco tem uma grande desvantagem: Caso um Pod seja recriado, apenas as camadas imutáveis do container serão recriadas, todo o resto será perdido[40]. Podemos então, fazer com que as mudanças feitas no disco do container sejam gravadas em uma pasta no sistema de arquivos do servidor. Com isto, a troca de containers não afetará os arquivos.

Chegamos então a um segundo problema: Um cluster Kubernetes pode conter dezenas, talvez até centenas de nodes[41]! Caso o Pod tenha que ser recriado em um servidor diferente, devemos garantir que os arquivos da pasta estejam disponíveis em todos os servidores em que este Pod possa ser criado.

Existem muitas soluções possíveis para este caso, desde criar pastas compartilhadas usando NFS[42] ou Samba[43] em todos os servidores do cluster ou mesmo, fazer com que Pods específicos sempre sejam executados em um mesmo servidor dedicado.

Para resolver este caso de maneira nativa do Kubernetes, podemos usar PV's e PVC's. Persistent Volumes e Persistent Volume Claims são recursos do Kubernetes usados para abstrair a camada de persistência de dados das aplicações. Os administradores do Cluster criam Volumes Persistentes, os PV's, que podem ser criados manualmente ou dinamicamente. Os Pods que precisam de Volumes Persistentes solicitam Persistent Volumes para o Cluster através de PVC's.

Caso o Cluster tenha um Volume Persistente livre, ele então o vincula ao PVC. Sempre que o Pod for iniciado em um servidor do cluster, o PV será iniciado em conjunto, sendo então montado no sistema de arquivos do servidor e do Pod.

Do ponto de vista da aplicação, tanto faz a tecnologia usada para criar o volume. Por trás dos panos, o Kubernetes pode ter solicitado que um provedor de nuvem crie um disco virtual para atender a solicitação, ou mesmo, que um

[39] https://aws.amazon.com/pt/s3/
[40] Union Filesystem
[41] Um node representa um servidores no Kubernetes.
[42] Network File System
[43] https://www.samba.org/samba/what_is_samba.html

disco seja montado usando iSCSI gerenciado por um HyperVisor[44].

9.1 StorageClass

Como dito antes, é possível criar Persistent Volumes de maneira dinâmica. Para isto usamos StorageClasses. Um StorageClass é uma forma de descrever um tipo, ou uma classe de volumes persistentes. Com isto, podemos por exemplo criar níveis de serviço em Storage, algo como uma classe para volumes que necessitem de alto desempenho e outra que priorize espaço em disco.

Podemos ver na figura 36 os StorageClasses instalados no ambiente através do comando: "kubectl get storageclass". StorageClasses existem fora de namespaces, por isso não precisamos definir nenhum no comando.

```
PS C:\Users\joao.arzamendia> kubectl get storageclass
NAME                 PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
standard (default)   rancher.io/local-path   Delete          WaitForFirstConsumer   false                  14h
```

Figura 36: Verificando os StorageClass do cluster.

Todo StorageClass deve ter os campos provisioner, parameters, e reclaimPolicy, que são usados para definir o comportamento de cada StorageClass, é muito importante notar que uma vez definidos, eles não podem mais ser alterados. O campo provisoner define o plugin usado para solicitar a criação da forma de storage em si dos PV's. Podemos ver um exemplo de StorageClass usado pelo Kind no código 9.1.

Código 13: Um StorageClass do Kind.

```
1  #storageClass.yaml
2  apiVersion: storage.k8s.io/v1                (1) Versao da API do Kubernetes
3  kind: StorageClass                           (2) Tipo do Recurso
4  metadata:
5    name: standard                             (3) Nome do Recurso
6    annotations: (4) Annotations que define como StorageClass padrao
7      storageclass.kubernetes.io/is-default-class: 'true'
8  provisioner: rancher.io/local-path       (5) Plugin de provisionamento
9  reclaimPolicy: Delete                     (6) Metodo usado ao deletar PVC
10 volumeBindingMode: WaitForFirstConsumer (7) Modo de Binding de volumes
```

Existem diversos plugins disponíveis por padrão. As versões do Kubernetes gerenciadas por provedores de Computação em Nuvem, como o AKS da Microsoft Azure, disponibiliza plugins que interagem com suas API's na criação de diversos tipos de discos virtuais, com performance e tamanho cobrados por demanda.

[44] Um gerenciador de máquinas virtuais.

9.2 Criando um PVC

Como dito antes, em versões gerenciadas do Kubernetes, a criação dinâmica de Persistent Volumes é feita através da criação de um PVC. O Kube-controller-manager irá iniciar um controlador que observará a criação de novos PVC's. Ele usará o StorageClass definido no PVC (ou o StorageClass padrão do ambiente caso nenhum seja definido) para criar PV's automaticamente.

No nosso caso, estamos usando o Kind para testar a criação de um PVC. Conforme podemos ver na 36, o StorageClass padrão do nosso cluster é Standard, que usa o provider rancher.io/local-path. A título de curiosidade, os PVs criados por este StorageClass são salvos como pastas no sistema de arquivos locais do node criado pelo Kind. Em nosso caso só há um servidor disponível, já que estamos em um ambiente controlado de testes.

Código 14: Um Persistent Volume Claim usando o storageClass standard.

```
1  #pvc.yaml
2  apiVersion: v1
3  kind: PersistentVolumeClaim
4  metadata:
5    name: pvc-kind
6    namespace: production
7  spec:
8    accessModes:
9      - ReadWriteOnce (1) Modo de acesso
10   resources:
11     requests:
12       storage: 10Gi  (2) Tamanha do PV
13   storageClassName: standard (3) StorageClass definido
14   volumeMode: Filesystem
```

Vamos começar criando o PVC descrito no código 9.2. Ele irá criar um PVC que solicitará um PV de 10 Gigaytes no modo de acesso ReadWriteOnce. Existem 3 tipos de modos de acesso: ReadWriteOnce, ReadOnlyMany e ReadWriteMany. No modo ReadWriteOnce o PV só pode ser acessado por um Pod por vez, em modo de leitura e escrita. No ReadOnlyMany, ele pode ser acessado por diversos Pods, porém em modo de apenas leitura. Já no modo ReadWriteMany, o PV poderá ser acessado por diversos Pods, em modo escrita e leitura.

Conforme a 37, usaremos o comando "kubectl apply -f ./storage/pvc/pvc.yaml -n production" para criar o PVC. Após isto, podemos descrever o PVC criado usando o comando "kubectl describe pvc pvc-kind -n production".

Como podemos ver na figura 38, o PVC está com o status Pending. Como nosso storageClass usa o volumeBindingMode do tipo WaitForFirstConsumer, ele irá criar o PV apenas quando um Pod solicitar o consumo do PVC. Existem

```
PS C:\git\apostila> kubectl apply -f .\storage\pvc\pvc.yaml -n production
persistentvolumeclaim/pvc-kind created
```

Figura 37: Criando um PVC.

storageClass que usam o modo Immediate, que cria o PV automaticamente também.

```
PS C:\git\apostila> kubectl describe pvc pvc-kind -n production
Name:          pvc-kind
Namespace:     production
StorageClass:  standard
Status:        Pending
Volume:
Labels:        <none>
Annotations:   <none>
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:
Access Modes:
VolumeMode:    Filesystem
Used By:       <none>
Events:
  Type    Reason               Age               From                         Message
  ----    ------               ----              ----                         -------
  Normal  WaitForFirstConsumer  12s (x4 over 45s)  persistentvolume-controller  waiting for first consumer to be created before binding
PS C:\git\apostila>
```

Figura 38: Um PVC em status pendente.

Agora então, para que o storageClass standard crie o PV, devemos criar um recurso que irá usar este PVC. Faremos isso com um recurso chamado StatefullSet.

Conforme podemos ver no código 9.2, temos aqui um recurso do Kubernetes feito para serviços com persistência de dados. Ele é muito parecido com um Deployment, porém é tratado de forma diferente. Uma das principais diferenças entre eles, é que os Pods criados por este recurso sempre terão o mesmo nome. Isso é feito para abarcar aplicações que precisem de mais estabilidade em seu ciclo de vida.

Em geral, aplicações que usem persistência em disco devem ser criadas usando StatefullSets. Com isto, estamos descrevendo de maneira correta o recurso para o Kubernetes. Com isto, ele terá mais informações sobre a aplicação quando for definir como e onde executá-la no cluster.

Código 15: Um StatefullSet que usa um PVC.

```
1 #statefullset .yaml
2 apiVersion: apps/v1        (1) Versao da API do Kubernetes
3 kind: StatefulSet          (2) Tipo do Recurso
4 metadata:
5   name: redis              (3) Nome do Recurso
6   namespace: production    (4) Namespace do Recurso
7 spec:
```

```
8   serviceName: "redis"      (5) Um Service criado para o recurso
9   selector:
10    matchLabels:
11      app: redis
12  updateStrategy:
13    type: RollingUpdate
14  replicas: 1
15  template:
16    metadata:
17      labels:
18        app: redis
19    spec:
20      containers:
21      - name: redis
22        image: redis
23        ports:
24          - containerPort: 6379
25        volumeMounts:                    (6) VolueMounts nos containers
26          - name: redis-data
27            mountPath: /usr/share/redis (7) Onde o PV sera montado
28      volumes:
29        - name: redis-data
30          persistentVolumeClaim:
31            claimName: pvc-kind      (8) PVC escolhido
```

Iremos usar o comando "kubectl create -f ./storage/pvc/statefullset.yaml -n production --save-config" para criar o StatefullSet. Na figura 39 podemos ver o procedimento.

```
PS C:\git\apostila> kubectl create -f .\storage\pvc\statefullset.yaml -n production --save-config
statefulset.apps/redis created
PS C:\git\apostila>
```

Figura 39: Criando um StatefullSet.

Após a criação do StatefullSet, um Pod será criado. Este Pod irá solicitar o PVC pvc-kind. Com isto um PV será criado e vinculado a este PVC. Podemos usar novamente o comando "kubectl describe pvc pvc-kind -n production" para ver a nova situação do PVC. Note o campo status e Volume. Conforme a imagem 40, o PVC está vinculado ao PV de nome pvc-8131a1f5-8fd4-468e-81b4-189bf25fa77d.

É muito importante notar que PVC's devem existir dentro de um Namespace (o mesmo do recurso que irá utilizá-lo). Ja PV's não são Namespaceds, não ficando vinculados a nenhum namespace.

```
PS C:\git\apostila> kubectl describe pvc pvc-kind -n production
Name:          pvc-kind
Namespace:     production
StorageClass:  standard
Status:        Bound
Volume:        pvc-8131a1f5-8fd4-468e-81b4-189bf25fa77d
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
               volume.beta.kubernetes.io/storage-provisioner: rancher.io/local-path
               volume.kubernetes.io/selected-node: kind-control-plane
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      10Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       redis-0
Events:
  Type    Reason                Age
  ----    ------                ----
  Normal  WaitForFirstConsumer  23s (x4 over 67s)
  Normal  ExternalProvisioning  20s (x2 over 20s)
ovisioner "rancher.io/local-path" or manually created by system administrator
  Normal  Provisioning          20s
oduction/pvc-kind"
  Normal  ProvisioningSucceeded  18s
b4-189bf25fa77d
```

Figura 40: Descrevendo um PVC que foi vinculado.

```
PS C:\git\apostila> kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   REASON   AGE
pvc-8131a1f5-8fd4-468e-81b4-189bf25fa77d   10Gi       RWO            Delete           Bound    production/pvc-kind   standard                6m30s
PS C:\git\apostila>
```

Figura 41: Verificando Persistent Volumes que criamos.

Podemos verificar todos os PV's criados para nosso cluster usando o comando: "kubectl get pv". Na imagem 41, podemos ver os PV's que acabamos de criar. Podemos descrevê-lo usando o comando "kubectl describe pv pvc-8131a1f5-8fd4-468e-81b4-189bf25fa77d". Na imagem 42 vemos o PV, notem o campo Source.Path que mostra o caminho no sistema de arquivos do servidor onde o volume foi criado.

Com isto, nossa aplicação não perderá os arquivos caso o container seja reiniciado. Um ponto importante a se pensar é que isto não garante o Backup dos arquivos em casos de recuperação de desastre ou perda de dados. Para isto, outras ferramentas devem ser usadas. Elas não estão presentes por padrão no Kubernetes.

```
PS C:\git\apostila> kubectl describe pv pvc-8131a1f5-8fd4-468e-81b4-189bf25fa77d
Name:              pvc-8131a1f5-8fd4-468e-81b4-189bf25fa77d
Labels:            <none>
Annotations:       pv.kubernetes.io/provisioned-by: rancher.io/local-path
Finalizers:        [kubernetes.io/pv-protection]
StorageClass:      standard
Status:            Bound
Claim:             production/pvc-kind
Reclaim Policy:    Delete
Access Modes:      RWO
VolumeMode:        Filesystem
Capacity:          10Gi
Node Affinity:
  Required Terms:
    Term 0:        kubernetes.io/hostname in [kind-control-plane]
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /var/local-path-provisioner/pvc-8131a1f5-8fd4-468e-81b4-189bf25fa77d_production_pvc-kind
    HostPathType:  DirectoryOrCreate
Events:            <none>
```

Figura 42: Verificando Persistent Volumes que criamos.

9.3 Conclusão

Neste capítulo nós tratamos da orquestração de Persistência de Dados no Kubernetes. Nós verificamos como a dinâmica entre Persistent Volume Claims e Persistent Volumes abstraem a forma como a persistência de dados será provida pelo administrador do cluster para as aplicações hospedadas em nosso ambiente.

Além disso, vimos como os StorageClass criam dinamicamente Persistent Volumes, através da criação de Persistent Volume Claims. Por fim, nós também vimos como descrever aplicações que usam Persistência de Disco usando StatefullSets vinculados a PVCs.

A manutenção de aplicações que mantém estado em disco é mais complexa e exige mais cuidados que aplicações sem estado em disco. É muito importante notar que o papel do Kubernetes, como em muitos casos, é apenas orquestrar de maneira correta os recursos da aplicação. A manutenção em si dos dados continua na mão dos operadores do ambiente.

10 ConfigMaps e Secrets

Aplicações precisam ser configuradas para funcionar corretamente. Isto pode ser feito de diversas maneiras. Podemos definir diretamente no código fonte da aplicação coisas como o endereço do banco de dados e as configurações de envio de e-mail. Podemos também usar arquivos de configuração ou mesmo variáveis de ambiente para configurar nossas aplicações.

As configurações também podem precisar ser alternadas conforme o ambiente em que estamos executando nossas aplicações. No ambiente de desenvolvimento poderíamos apontar a aplicação para um banco de dados local e no ambiente de produção, um banco de dados hospedado mais poderoso na nuvem. Também podemos ter diferentes configurações para o ambiente de produção e homologação.

Para facilitar este tipo de coisa, o Kubernetes oferece um recurso chamado ConfigMap. ConfigMaps são recursos responsáveis por descrever arquivos de configuração que podem ser injetados em Pods. Com eles, podemos compartilhar configurações entre diversos Pods ou mesmo diferenciar ambientes através da escolha do ConfigMap correto para cada um deles. Podemos criar grupos de arquivos de configuração e injetá-los em uma pasta dentro de um Pod ou mesmo como variáveis de ambiente do container.

Código 16: Um ConfigMap baseado em um arquivo.

```
1  #configmap.yaml
2  apiVersion: v1                          (1) API do Kubernetes usada
3  kind: ConfigMap                         (2) Tipo do recurso
4  metadata:
5    name: nginx-conf                      (3) Nome do recurso
6  data:                                   (4) Inicio do bloco de configuracoes
7    nginx.conf: |                         (5) Nome do arquivo
8      user nginx;                         (6) Inicio do conteudo do arquivo
9      worker_processes 1;
10     events {
11       worker_connections 10240;
12     }
13     http {
14       server {
15           listen       80;
16           server_name  localhost;
17           location / {
18             root   /usr/share/nginx/html;
19             index  index.html index.htm;
20         }
21       }
22     }
```

Quando o conteúdo de um arquivo de configuração é sensível, como uma senha de banco de dados ou uma chave de acesso a algum recurso, devemos dar a ele um tratamento especial. Para isto o Kubernetes tem um recurso chamado Secret.

Secrets, assim como ConfigMaps, servem para injetarmos configurações dentro de Pods. A diferença é que o Kubernetes trata as secrets de maneira mais segura. Por padrão as secrets são salvas no formato Base64 e é possível configurar o Kubernetes para encriptar seu conteúdo ao salvá-las no ETCD.

Além disso, é possível integrar o Kubernetes a serviços, em geral conhecidos como Cofres de Senhas, onde as Secrets serão gerenciadas externamente ao ambiente. Independente da forma como elas forem gerenciadas, é uma boa prática utilizar as Secrets do Kubernetes como forma de injetar credenciais em Pods.

Secrets também podem ser utilizadas para salvar certificados digitais em um formato adequado para integração em diversos recursos do Kubernetes, como por exemplo para definir o certificado digital de um site usando um Ingress Controller. Também é possível injetar credenciais para Registros de Containers privados através de Secrets, aplicando-as diretamente a Pods.

Código 17: Uma Secret básica.

```
1 #secret.yaml
2 apiVersion: v1                          (1) API do Kubernetes usada
3 kind: Secret                            (2) Tipo do recurso
4 metadata:
5   name: secret-basic-auth               (3) Nome do recurso
6 type: kubernetes.io/basic-auth          (4) Tipo da Secret
7 stringData:                             (5) Inicio  do bloco de dados
8   username: admin                       (6) Usuario
9   password: t0p-Secret                  (7) Senha
```

Secrets podem ser criados de diversas formas e podem ser usados em várias funções. Os principais tipos são:

- Opaque
- kubernetes.io/dockerconfigjson
- kubernetes.io/basic-auth
- kubernetes.io/tls

As Secrets do tipo Opaque podem receber qualquer conteúdo escolhido pelo usuário, de forma parecida com um ConfigMap. As Secrets do tipo dockercon-

figjson mantém o arquivo config.json do Docker. É muito comum usá-la para manter as credenciais de acesso a um registro de containers.

Secrets do tipo basic-auth matem duplas de usuários e senhas, que podem ser usados de forma fácil em recursos que aceitem esse tipo de configuração e por fim os do tipo kubernetes.io/tls, que mantém certificados digitais.

10.1 Criando ConfigMaps

Podemos criar ConfigMaps diretamente através de um arquivo de configuração já existente. Para isto, podemos usar o comando "kubectl create configmap nginx-conf --from-file=./configmaps/nginx.conf -n production", que com base no arquivo nginx.conf irá criar o ConfigMap nginx-conf no Namespace production. Na figura 43 podemos observar o procedimento:

```
PS C:\git\apostila> kubectl create configmap nginx-conf --from-file=./configmaps/nginx.conf -n production
configmap/nginx-conf created
```

Figura 43: Criando um ConfigMap através de um arquivo de configuração.

Uma vez criado, podemos injetar um ConfigMap através de VolumeMounts em um Deployment, assim garantindo que todos os Pods da aplicação tenham acesso ao arquivo. Para isto podemos usar o seguinte Deployment definido em 10.1:

Código 18: Deployment com ConfigMap.

```
1  #deployment_cm.yaml
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: nginx-deployment-cm
6    namespace: production
7    labels:
8      app: nginx
9  spec:
10   replicas: 1
11   selector:
12     matchLabels:
13       app: nginx
14   template:
15     metadata:
16       labels:
17         app: nginx
18     spec:
```

```
19      containers:
20      - name: nginx
21        image: nginx:1.14.2
22        ports:
23        - containerPort: 80
24        volumeMounts:                (5) Montamos os Volumes
25        - name: nginx-conf           (7) Nome do Volume definido em 2
26          mountPath: /config         (8) Pasta onde o volume sera montado
27      volumes:                       (1) Definimos os Volumes
28        - name: nginx-conf           (2) Definimos o nome do volume
29          configMap:                 (3) Volume baseado em ConfigMap
30            name: nginx-conf         (4) Nome do ConfigMap
```

Como vemos na figura 44, usaremos o seguinte comando para criar o Deployment no mesmo Namespace do ConfigMap: "kubectl apply -f ./deployments/deployment_cm.yaml -n production"

```
PS C:\git\apostila> kubectl apply -f ./deployments/deployment_cm.yaml -n production
deployment.apps/nginx-deployment-cm created
```

Figura 44: Criando um Deployment que use um ConfigMap.

Após isto, como vemos na figura 45, podemos descrever o Deployment recém criado para verificar se tudo deu certo. Faremos isso com o comando "kubectl describe deployment nginx-deployment-cm -n production":

Por fim, devemos identificar os Pods do nosso Deployment através do comando "kubectl get pods -n production", e com seu nome, verificar se o arquivo de configuração foi realmente adicionado a pasta /conf. Para isto, usaremos o comando "kubectl exec -it pod/nginx-deployment-cm-665dd658c7-sx7b2 -n production -- cat /config/nginx.conf", na figura 46:

ConfigMaps são muito uteis em aplicações que mantém um mesmo código para diferentes ambientes, modificando apenas arquivos de configuração para cada um deles. Com isto, podemos orquestrar as configurações das nossas aplicações através do Kubernetes.

10.2 Criando Secrets

Podemos criar Secrets através da linha de comando ou de arquivos YAML. Várias Secrets são criadas automaticamente pelo Kubernetes, já que coisas como os tokens gerados para ServiceAccounts[45] ou configurações sensíveis do Kuber-

[45] Service Accounts são vinculadas ao RBAC, o controle de acesso aos recursos do Kubernetes.

```
PS C:\git\apostila> kubectl describe configmap nginx-conf -n production
Name:         nginx-conf
Namespace:    production
Labels:       <none>
Annotations:  <none>

Data
====
nginx.conf:
----
user nginx;
worker_processes  1;
events {
    worker_connections  10240;
}
http {
    server {
        listen       80;
        server_name  localhost;
        location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
    }
}
Events:  <none>
```

Figura 45: Descrevendo um Deployment que usa ConfigMaps.

netes são em geral, salvas como Secrets.

10.2.1 Criando a partir de Certificados

Começaremos pela criação de uma Secret baseada em um certificado digital. Inicialmente vamos criar um certificado auto-assinado, para isto usaremos o Openssl[46]. Com o comando: "openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt -subj '/CN=www.ufms.br'" criaremos um conjunto de certificado e chave. Na figura 47 podemos observar o procedimento:

Após isto, como vemos na figura 43,podemos criar a Secret através do comando "kubectl create secret tls certificado --key='tls.key' --cert='tls.crt' -n production". Com a Secret criada podemos descrevê-la com o comando: "kubectl describe secret certificado -n production".

Através do comando "kubectl get secret certificado -n production -o yaml"

[46] https://www.openssl.org/

Figura 46: Verificando se um arquivo de configuração foi criado.

Figura 47: Criando um certificado com o Openssl.

podemos ter uma visão mais completa do seu conteúdo. na figura 49 podemos ver o conteúdo do certificado.

10.2.2 Criando senhas de acesso a Registros

Em casos onde as imagens usadas para a criação dos Pods estejam em repositórios privados, credenciais de acesso devem ser usadas ao fazer o Pull das imagens.

Para isto, podemos criar estas credenciais como Secrets para depois adicioná-las como uma configuração descrita no YAML do Pod, a imagePullSecrets.

Para criar uma imagePullSecrets, usaremos o comando: "kubectl create secret docker-registry pull-secret --docker-username=user --docker-password=pass --docker-email=email -n production".

Após criar a Secret, como vemos na figura 50, podemos descrevê-la com o comando: "kubectl describe secret pull-secret -n production".

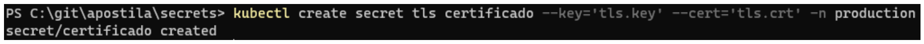

Figura 48: Criando uma Secret com um Certificado Digital.

Figura 49: O YAML de um Secret.

10.3 Conclusão

ConfigMaps e Secrets são recursos muito utilizados por aplicações criadas pensando no Kubernetes. Eles são formas de abstrair configurações que poderiam se tornar extremamente complexas em ambientes onde não temos total controle dos servidores que nossas aplicações estão hospedadas. Com isto, aplicações pensadas para Orquestradores de Containers ou Computação em Nuvem ganham muito com a adição de configuração orquestrada.

As aplicações de apoio de um cluster Kubernetes, como um Ingress Controller, em geral sempre usam ConfigMaps e Secrets seja para configuração ou customização de seus recursos.

Com ConfigMaps conseguimos facilitar o uso de aplicações imutáveis, onde as mesmas imagens de containers podem ser usados tanto para o ambiente de produção ou homologação, com eles sendo diferenciados apenas pelos ConfigMaps e Secrets definidos em seus Deployments.

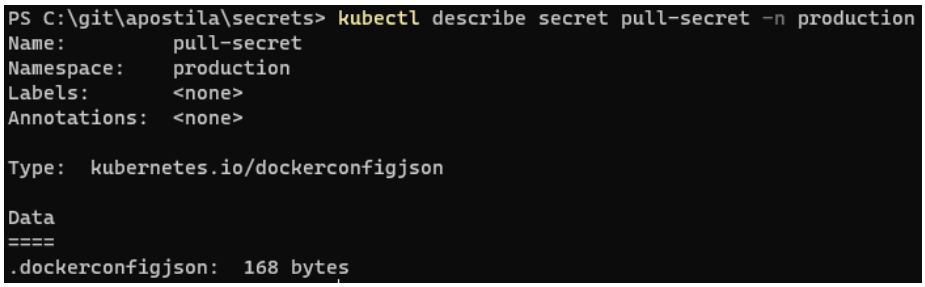

Figura 50: Criando uma Secret para registros privados.

11 Conclusão

No decorrer deste documento passamos por vários recursos do Kubernetes. Abordamos suas funcionalidades e formas de uso. Criamos esses recursos em um ambiente controlado para demonstrar suas funções básicas. Com isto, devemos ter percorrido cerca de 1% de tudo o que é possível de se fazer no Kubernetes.

Tirando o Ingress Controller, abordamos apenas os recursos nativos do Kubernetes. Fora eles, temos centenas de outros recursos customizados que podem ser integrados em nossos clusters, que adicionam funções extremamente úteis em nossos ambientes de produção.

No site https://landscape.cncf.io/ da CNCF, que desde 2016 é a dona do Kubernetes, estão listados atualmente 1077 plataformas e aplicações nativas na nuvem de diversas categorias, em sua maioria feitas para serem usadas no Kubernetes!

As possibilidades são imensas, o Kubernetes é um projeto em constante evolução e justamente por isto, fica a nosso cargo está constantemente se atualizando e adaptando-se às novas ferramentas e recursos que o Kubernetes nos entrega.