Aula 23 – Principais Objetos do Kubernetes – Parte 2

No universo das arquiteturas de aplicações web modernas, a orquestração de contêineres se tornou um pilar fundamental para garantir escalabilidade, resiliência e agilidade. O Kubernetes, nesse cenário, emerge como a ferramenta dominante, oferecendo um ecossistema robusto para gerenciar cargas de trabalho complexas. Contudo, para realmente dominar o Kubernetes, é essencial ir além dos conceitos básicos e mergulhar nos objetos que permitem a construção de aplicações verdadeiramente dinâmicas e seguras.

Esta aula é a continuação de nossa jornada pelos objetos essenciais do Kubernetes, aprofundando-se em componentes que resolvem desafios cruciais no dia a dia do desenvolvimento e operação de sistemas distribuídos. Se na parte anterior exploramos os fundamentos, agora vamos desvendar como gerenciar configurações e dados sensíveis, expor suas aplicações ao mundo externo de forma inteligente e garantir que seus dados persistam, independentemente do ciclo de vida dos seus contêineres.

Ao final desta aula, você será capaz de compreender a necessidade e a aplicação prática de ConfigMaps e Secrets para gerenciar configurações e dados sensíveis de forma eficiente e segura. Além disso, entenderá como o Ingress facilita a exposição de serviços HTTP/HTTPS, e como PersistentVolumes e PersistentVolumeClaims garantem a persistência de dados em um ambiente dinâmico. Prepare-se para elevar suas habilidades em Kubernetes e construir aplicações ainda mais robustas e preparadas para os desafios do futuro.

Gerenciando Configurações com ConfigMaps

Imagine que você está desenvolvendo uma aplicação e precisa que ela se comporte de maneiras diferentes em ambientes distintos – por exemplo, usando um banco de dados de desenvolvimento localmente e um banco de dados de produção na nuvem. Ou talvez você queira ativar ou desativar uma funcionalidade específica sem precisar recompilar e reimplantar todo o seu código. Tradicionalmente, essas configurações poderiam ser embutidas no código ou passadas como variáveis de ambiente, mas em um ambiente de contêineres dinâmico como o Kubernetes, essas abordagens rapidamente se tornam um pesadelo de gerenciamento.

O problema central é como desacoplar a configuração do código da aplicação, permitindo que ela seja alterada e aplicada sem interrupção ou reconstrução.

É aqui que os **ConfigMaps** entram em cena, oferecendo uma solução elegante e padronizada para injetar dados de configuração em seus Pods. Eles atuam como um repositório centralizado de pares chave-valor, onde as chaves são nomes de configuração e os valores são os dados que sua aplicação precisa.

Pense nos ConfigMaps como um "livro de receitas" para suas aplicações. Assim como um chef consulta um livro de receitas para saber a quantidade exata de sal ou o tempo de cozimento para um prato, sua aplicação consulta um ConfigMap para obter as instruções sobre como se conectar a um serviço externo, qual nível de log usar ou quais recursos habilitar. Essa abordagem permite que o mesmo código da aplicação seja usado em diferentes ambientes, bastando alterar o ConfigMap associado.

Flexibilidade

Altere configurações sem recompilar código

Centralização

Gerencie todas as configs em um único lugar

Reutilização

Mesmo código em múltiplos ambientes

Um exemplo prático seria uma aplicação que precisa de uma URL de API externa e um nível de log. Em vez de codificar esses valores, você criaria um ConfigMap com chaves como API_URL e LOG_LEVEL. Seu Pod, então, seria configurado para ler esses valores do ConfigMap, seja como variáveis de ambiente ou como arquivos montados dentro do contêiner. Isso significa que, se a URL da API mudar ou você precisar de logs mais detalhados, basta atualizar o ConfigMap e reiniciar os Pods (ou usar estratégias de atualização mais avançadas), sem tocar no código da aplicação. Essa flexibilidade é crucial para a agilidade e a manutenção em ambientes de microserviços.

ConfigMaps em Ação: Criando e Consumindo

A criação de um ConfigMap é bastante direta, geralmente feita através de um arquivo YAML. Você pode definir os dados de configuração diretamente no YAML ou referenciar arquivos externos. Uma vez criado, o ConfigMap se torna um recurso disponível no seu cluster Kubernetes, pronto para ser consumido pelos seus Pods. A beleza reside na simplicidade e na capacidade de centralizar informações que, de outra forma, estariam espalhadas ou embutidas.

Duas Formas de Consumir ConfigMaps

Variáveis de Ambiente

Ideais para valores simples e únicos, como um nome de banco de dados ou uma porta de conexão.

- Fácil de implementar
- Acesso direto no código
- Perfeito para configs simples

Volumes Montados

Perfeitos para arquivos de configuração complexos, como application.properties ou nginx.conf.

- Suporta arquivos completos
- Leitura como arquivo local
- Ideal para configs complexas

Exemplo Prático: Aplicação Node.js

Considere uma aplicação Node.js que precisa de uma porta para escutar e uma mensagem de boas-vindas. Poderíamos ter um ConfigMap chamado minha-app-config com os seguintes dados:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: minha-app-config
data:
  PORTA_APP: "3000"
  MENSAGEM_BOAS_VINDAS: "Bem-vindo ao meu serviço!"
```

Para que um Pod consuma esses dados como variáveis de ambiente, a definição do Pod incluiria algo como:

```
apiVersion: v1
kind: Pod
metadata:
  name: minha-app-pod
spec:
  containers:
  - name: minha-app-container
    image: minha-imagem-node:latest
    env:
    - name: APP_PORT
      valueFrom:
        configMapKeyRef:
          name: minha-app-config
          key: PORTA_APP
    - name: WELCOME_MESSAGE
      valueFrom:
        configMapKeyRef:
          name: minha-app-config
          key: MENSAGEM_BOAS_VINDAS
```

Resultado: A lógica da aplicação permanece limpa e as configurações podem ser gerenciadas e atualizadas de forma independente. É um passo fundamental para construir aplicações verdadeiramente nativas da nuvem.

Desafios e Boas Práticas com ConfigMaps

Embora os ConfigMaps sejam poderosos para gerenciar configurações, é crucial entender suas limitações e aplicar boas práticas. Uma das principais considerações é que os ConfigMaps não são projetados para armazenar dados sensíveis. Eles são armazenados sem criptografia no etcd (o banco de dados do Kubernetes) e podem ser facilmente acessados por qualquer pessoa com permissões para ler ConfigMaps no cluster. Para dados como senhas, chaves de API ou tokens, o Kubernetes oferece um objeto específico que veremos a seguir.

Limitação Importante

ConfigMaps **não são criptografados** e não devem conter dados sensíveis como senhas ou tokens de API.

Propagação de Atualizações

Pods com variáveis de ambiente não veem mudanças automaticamente. Volumes montados podem levar alguns minutos para atualizar.

Boas Práticas Essenciais

01

Granularidade

Crie ConfigMaps menores e mais específicos para diferentes componentes ou funcionalidades, em vez de um ConfigMap gigante para toda a aplicação. Isso facilita a manutenção e evita que mudanças em uma parte afetem outras.

02

Versionamento

Embora o Kubernetes não versionamento ConfigMaps nativamente, você pode usar ferramentas externas ou convenções de nomenclatura (ex: minha-app-config-v2) para gerenciar diferentes versões.

03

Ambientes

Utilize ConfigMaps distintos para diferentes ambientes (desenvolvimento, staging, produção) para garantir que cada um tenha suas configurações específicas.

04

Não sensível

Lembre-se sempre: ConfigMaps são para dados não sensíveis. Qualquer informação que possa comprometer a segurança da sua aplicação ou dos seus usuários deve ser tratada de forma diferente.

Ao seguir essas diretrizes, você maximiza os benefícios dos ConfigMaps, tornando suas aplicações mais flexíveis, fáceis de gerenciar e prontas para escalar em um ambiente Kubernetes.

Lidando com Dados Sensíveis: Os Secrets do Kubernetes

Após entender como gerenciar configurações gerais com ConfigMaps, surge uma questão crítica: e quanto aos dados que não podem ser expostos abertamente? Senhas de banco de dados, chaves de API de serviços externos, certificados TLS e outros tokens de autenticação são informações altamente sensíveis que, se comprometidas, podem levar a sérias violações de segurança. Armazená-los em ConfigMaps seria um risco inaceitável, pois eles não oferecem nenhuma camada de proteção intrínseca.

O Kubernetes aborda essa necessidade com os Secrets. Um Secret é um objeto do Kubernetes projetado especificamente para armazenar e gerenciar informações sensíveis, como senhas, tokens OAuth e chaves SSH.

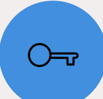
Ele oferece um mecanismo mais seguro do que ConfigMaps para distribuir essas credenciais para os Pods, embora seja importante notar que, por padrão, os Secrets são apenas codificados em Base64, não criptografados no etcd. A verdadeira segurança vem das permissões de acesso (RBAC) e, idealmente, da integração com sistemas de gerenciamento de chaves externos (KMS).

Proteção de Credenciais

Senhas, tokens e chaves de API protegidos

Controle de Acesso

RBAC garante acesso apenas autorizado



Integração KMS

Criptografia adicional com serviços externos

Pense nos Secrets como um "cofre de segurança" dentro do seu cluster Kubernetes. Assim como você guardaria seus documentos mais importantes e joias em um cofre bancário, os Secrets guardam as credenciais mais valiosas da sua aplicação. Embora o cofre em si não seja invisível, o acesso a ele é estritamente controlado, e o conteúdo dentro dele é protegido contra olhares curiosos. A ideia é que apenas quem tem a chave (as permissões corretas) pode abrir e ver o que está dentro.

Um exemplo clássico é uma aplicação que precisa se conectar a um banco de dados. Em vez de colocar o nome de usuário e a senha diretamente no código ou em um ConfigMap, você criaria um Secret contendo essas credenciais. O Pod da sua aplicação seria então configurado para montar esse Secret como um volume ou injetá-lo como variáveis de ambiente. Dessa forma, as credenciais não ficam expostas no código-fonte, nos logs ou em arquivos de configuração abertos, reduzindo significativamente a superfície de ataque.

Secrets em Ação: Protegendo Suas Credenciais

A criação de um Secret é similar à de um ConfigMap, mas com a particularidade de que os valores dos dados devem ser codificados em Base64. Isso não é uma criptografia, mas uma codificação que garante que os dados sejam tratados como texto seguro e não causem problemas de parsing em YAML ou JSON. Para uma segurança robusta, o Kubernetes permite a integração com provedores de KMS (Key Management Service) externos, que podem criptografar os Secrets no etcd.

Criando um Secret

Um Secret pode ser criado a partir de arquivos literais, de um arquivo YAML ou usando o comando kubectl create secret. Por exemplo, para criar um Secret com um nome de usuário e senha para um banco de dados:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque  # Tipo genérico de Secret
data:
  username: YWRtaW4=  # 'admin' em Base64
  password: c2VuaGFzZWNyZXRh  # 'senhasecreta' em Base64
```

Consumindo um Secret em um Pod

Para consumir este Secret em um Pod, a abordagem é muito parecida com a dos ConfigMaps, mas com a diferença que os dados do Secret são montados em um diretório temporário e seguro dentro do contêiner.

```
apiVersion: v1
kind: Pod
metadata:
  name: minha-app-db-pod
spec:
  containers:
  - name: minha-app-db-container
    image: minha-imagem-app:latest
    env:
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: username
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: password
```

Montagem como Volume

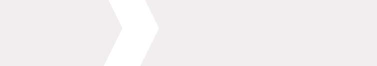
Alternativamente, você pode montar o Secret como um volume, onde cada chave do Secret se torna um arquivo dentro do diretório montado. Isso é útil para certificados TLS ou arquivos de chave.

```
volumeMounts:
- name: db-secret-volume
  mountPath: "/etc/secrets/db"
  readOnly: true
volumes:
- name: db-secret-volume
  secret:
    secretName: db-credentials
```

Resultado: Essa flexibilidade permite que as aplicações acessem as credenciais de forma padronizada, sem que elas precisem ser expostas em locais inseguros. A gestão de Secrets é um pilar da segurança em ambientes de contêineres.

Boas Práticas e Segurança com Secrets

A segurança dos Secrets no Kubernetes é um tópico complexo e de extrema importância. Embora os Secrets ofereçam um mecanismo para gerenciar dados sensíveis, eles não são uma solução "plug-and-play" para todas as preocupações de segurança. É fundamental adotar uma série de boas práticas para garantir que suas credenciais permaneçam protegidas.



Base64 ≠ Criptografia

A codificação Base64 não é criptografia. Qualquer pessoa com acesso ao etcd pode decodificar facilmente os valores.

RBAC é Essencial

Utilize controle de acesso baseado em função para limitar rigorosamente quem pode criar, visualizar e modificar Secrets.

Integração KMS

Use serviços externos de gerenciamento de chaves para criptografar Secrets no etcd, adicionando camada extra de segurança.

Recomendações de Segurança Avançada

- **Controle de Acesso Rigoroso:** Utilize o RBAC (Role-Based Access Control) do Kubernetes para limitar rigorosamente quem pode criar, visualizar e modificar Secrets. Apenas os Pods que realmente precisam de um Secret específico devem ter permissão para acessá-lo.
- **KMS Externo:** Considere a integração com um KMS (Key Management Service) externo. Soluções como HashiCorp Vault, AWS KMS, Azure Key Vault ou Google Cloud KMS podem ser usadas para criptografar os Secrets no etcd, adicionando uma camada extra de segurança. Isso significa que, mesmo que o etcd seja comprometido, os dados dos Secrets permanecerão criptografados e ilegíveis sem a chave do KMS.
- **Nunca no Git:** Evite armazenar Secrets em repositórios de código-fonte (Git). Mesmo que estejam codificados em Base64, eles ainda são facilmente decodificáveis. Use ferramentas de CI/CD para injetar Secrets no cluster de forma segura durante o processo de deploy, ou utilize soluções como o Sealed Secrets.

Quadro Comparativo: ConfigMaps vs. Secrets

Característica	ConfigMaps	Secrets
Propósito	Gerenciar dados de configuração não sensíveis	Gerenciar dados sensíveis (senhas, chaves, etc.)
Segurança	Armazenados em texto plano (ou Base64)	Armazenados em Base64 (não criptografado por padrão)
Acesso	Controlado por RBAC	Controlado por RBAC (mais restritivo)
Uso Comum	URLs de API, níveis de log, feature flags	Credenciais de DB, chaves de API, certificados TLS
Recomendação	Para dados públicos ou semi-públicos	Para dados que exigem proteção rigorosa

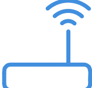
Ao seguir essas diretrizes e entender a distinção clara entre ConfigMaps e Secrets, você estará construindo aplicações mais seguras e robustas no Kubernetes, protegendo os ativos mais valiosos da sua infraestrutura.

Expondo Serviços ao Mundo Externo com Ingress

Até agora, vimos como gerenciar configurações e dados sensíveis dentro do cluster Kubernetes. Mas como os usuários externos acessam suas aplicações? Por padrão, os Services do Kubernetes (como ClusterIP) são acessíveis apenas de dentro do cluster. Para expor um Service, você pode usar um Service do tipo NodePort ou LoadBalancer. No entanto, essas opções têm suas limitações, especialmente quando você precisa de roteamento baseado em nome de host ou caminho, terminação SSL/TLS, ou um único ponto de entrada para múltiplos serviços.

> **É nesse ponto que o Ingress se torna indispensável.** O Ingress é um objeto API do Kubernetes que gerencia o acesso externo a serviços dentro do cluster, tipicamente HTTP e HTTPS.

Ele atua como um roteador de tráfego inteligente, permitindo que você defina regras para direcionar requisições de entrada para diferentes Services com base no nome do host, no caminho da URL ou em outras condições.



Roteamento Inteligente

Direcione tráfego baseado em host, caminho ou outras condições para diferentes serviços de forma organizada.

Terminação SSL/TLS

Gerencie certificados e descriptografe tráfego HTTPS em um único ponto centralizado.

Ponto de Entrada Único

Um único IP externo para múltiplos serviços, simplificando a gestão de tráfego.

Imagine o Ingress como o "recepcionista" de um grande hotel ou o "controlador de tráfego" de uma cidade movimentada. Quando um visitante (requisição externa) chega, o recepcionista (Ingress) não o envia para um quarto aleatório. Em vez disso, ele verifica o nome do visitante (nome do host ou caminho da URL) e o direciona para o andar e quarto corretos (o Service apropriado). Isso permite que um único ponto de entrada gerencie o fluxo para muitos destinos diferentes, de forma organizada e eficiente.

Um exemplo prático seria ter uma aplicação com um frontend web e um backend de API. Você pode querer que minhaapp.com aponte para o serviço do frontend e minhaapp.com/api aponte para o serviço do backend. Sem Ingress, você precisaria de dois LoadBalancers separados ou de uma configuração complexa em um proxy externo. Com Ingress, você define essas regras de roteamento de forma declarativa dentro do Kubernetes, e um Ingress Controller (como Nginx Ingress Controller ou Traefik) se encarrega de implementá-las.

Ingress em Ação: Roteamento Inteligente e SSL

O Ingress não é um serviço em si, mas uma coleção de regras. Para que essas regras funcionem, você precisa de um **Ingress Controller** rodando no seu cluster. O Ingress Controller é uma aplicação que monitora o recurso Ingress e configura um proxy reverso (como Nginx, HAProxy, ou um Load Balancer de nuvem) para rotear o tráfego de acordo com as regras definidas.

Definindo um Ingress

A definição de um Ingress é feita através de um arquivo YAML, onde você especifica as regras de roteamento, os hosts, os caminhos e os Services de destino. Além disso, o Ingress pode lidar com a terminação SSL/TLS, o que significa que ele pode descriptografar o tráfego HTTPS antes de encaminhá-lo para os Services internos, simplificando a gestão de certificados.

Considere um cenário onde você tem dois serviços: frontend-service na porta 80 e backend-service na porta 8080. Você quer que app.exemplo.com vá para o frontend e app.exemplo.com/api vá para o backend.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minha-app-ingress
spec:
  rules:
  - host: app.exemplo.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 8080
```

Este Ingress direcionaria o tráfego para app.exemplo.com/ para o frontend-service e para app.exemplo.com/api para o backend-service.

Adicionando SSL/TLS

Para adicionar SSL/TLS, você precisaria de um Secret contendo o certificado e a chave privada, e então referenciá-lo no Ingress:

```
spec:
  tls:
  - hosts:
    - app.exemplo.com
    secretName: app-tls-secret  # Secret contendo o certificado TLS
  rules:
  # ... (regras de host e path como acima)
```

Resultado: Essa capacidade de roteamento avançado e gerenciamento de SSL/TLS centralizado torna o Ingress uma peça fundamental para expor aplicações complexas no Kubernetes, oferecendo flexibilidade e eficiência que as opções de Service mais simples não conseguem.

Ingress vs. Outros Tipos de Service

Entender a diferença entre Ingress e os tipos de Service como NodePort e LoadBalancer é crucial para escolher a estratégia de exposição correta para suas aplicações. Embora todos eles permitam que o tráfego externo chegue aos seus Pods, eles operam em diferentes camadas e oferecem capacidades distintas.

NodePort

Um **Service do tipo NodePort** expõe um Service em uma porta estática em cada nó do cluster. Isso significa que você pode acessar o Service usando o IP de qualquer nó e a porta NodePort.

- Simples de configurar
- Expõe portas nos nós
- Sem balanceamento avançado
- Ideal para desenvolvimento

LoadBalancer

Um **Service do tipo LoadBalancer** provisiona um balanceador de carga externo (se o seu provedor de nuvem suportar) que roteia o tráfego para os nós do seu cluster.

- IP externo estável
- Balanceamento básico
- Um IP por Service
- Pode ser custoso

Ingress

O **Ingress** não é um tipo de Service, mas um objeto que funciona em conjunto com um Ingress Controller. Opera na camada 7 (HTTP/HTTPS).

- Roteamento avançado
- Terminação SSL/TLS
- Um IP para múltiplos Services
- Ideal para produção

Recursos Avançados do Ingress

Roteamento baseado em host

app1.exemplo.com para Service A, app2.exemplo.com para Service B.

Roteamento baseado em caminho

exemplo.com/api para Service A, exemplo.com/dashboard para Service B.

Terminação SSL/TLS

Gerencia certificados e descriptografa o tráfego HTTPS.

Balanceamento de carga

O Ingress Controller pode oferecer algoritmos de balanceamento de carga mais avançados.

Um único IP externo

Vários Services podem ser expostos através de um único endereço IP público do Ingress Controller.

Quadro Comparativo: Tipos de Exposição de Serviços

Característica	NodePort	LoadBalancer	Ingress
Camada	Camada 4 (TCP/UDP)	Camada 4 (TCP/UDP)	Camada 7 (HTTP/HTTPS)
IP Externo	IP dos nós + porta estática	IP externo provisionado pelo provedor de nuvem	IP do Ingress Controller (geralmente LoadBalancer)
Roteamento	Básico (para um único Service)	Básico (para um único Service)	Avançado (host, caminho, etc.)
SSL/TLS	Não nativo (precisa de configuração no Service)	Não nativo (precisa de configuração no Service)	Nativo (terminação SSL)
Custo	Baixo	Moderado (custo do LoadBalancer)	Moderado (custo do LoadBalancer do Ingress Controller)
Uso Ideal	Desenvolvimento, testes, serviços internos	Exposição simples de um único Service	Exposição complexa de múltiplos Services, roteamento avançado

A escolha do método de exposição depende das suas necessidades. Para a maioria das aplicações web em produção, o Ingress é a opção preferida devido à sua flexibilidade e recursos avançados, permitindo uma gestão centralizada e eficiente do tráfego de entrada.

Gerenciando Armazenamento Persistente com PersistentVolumes

Um dos princípios fundamentais dos contêineres é a efemeridade: eles são projetados para serem descartáveis e sem estado. Isso é ótimo para escalabilidade e resiliência, mas apresenta um desafio significativo para aplicações que precisam armazenar dados de forma duradoura, como bancos de dados, sistemas de arquivos ou caches persistentes. Se um Pod é reiniciado ou realocado para outro nó, todos os dados armazenados dentro do contêiner são perdidos.

Para resolver esse problema, o Kubernetes introduz o conceito de **PersistentVolumes (PVs)** e **PersistentVolumeClaims (PVCs)**.

Um PersistentVolume é um pedaço de armazenamento no cluster que foi provisionado por um administrador ou dinamicamente por um StorageClass. Ele é um recurso de cluster, independente do ciclo de vida de um Pod. Já um PersistentVolumeClaim é uma requisição de armazenamento feita por um usuário (ou por uma aplicação em um Pod). É como um Pod que requisita recursos de CPU e memória, mas para armazenamento.

PersistentVolume (PV)

Recurso de armazenamento provisionado no cluster, independente do ciclo de vida dos Pods. Representa o armazenamento físico disponível.

PersistentVolumeClaim (PVC)

Requisição de armazenamento feita por um usuário ou aplicação. Especifica necessidades como tamanho e modo de acesso.

Pense nos PersistentVolumes como "terrenos" ou "espaços de armazenamento" disponíveis para aluguel em uma grande propriedade (o cluster). O administrador do cluster prepara esses terrenos, definindo seu tamanho, tipo (SSD, HDD) e como eles podem ser acessados (NFS, iSCSI, provedores de nuvem). Quando uma aplicação precisa de armazenamento, ela não se preocupa com os detalhes do terreno em si, mas apenas faz um "pedido de aluguel" (PersistentVolumeClaim) especificando suas necessidades (tamanho, modo de acesso). O Kubernetes, então, encontra um terreno disponível que atenda a esses requisitos e o "aluga" para a aplicação.

Essa abstração é poderosa porque desacopla a forma como o armazenamento é provisionado da forma como ele é consumido. Os desenvolvedores de aplicações podem simplesmente solicitar armazenamento sem se preocupar com a infraestrutura subjacente, enquanto os administradores podem gerenciar o armazenamento de forma centralizada e flexível. Isso é essencial para rodar aplicações stateful (com estado) no Kubernetes, garantindo que seus dados permaneçam seguros e acessíveis, mesmo quando os Pods vêm e vão.

PersistentVolumes e PersistentVolumeClaims em Ação

A criação e o consumo de armazenamento persistente no Kubernetes envolvem dois passos principais: primeiro, o provisionamento de um PersistentVolume (ou a configuração de um StorageClass para provisionamento dinâmico); segundo, a criação de um PersistentVolumeClaim para solicitar esse armazenamento.

Criando um PersistentVolume (PV)

Um **PersistentVolume (PV)** é definido por um administrador e representa o armazenamento físico. Ele pode ser um volume NFS, um disco de nuvem (AWS EBS, Google Persistent Disk, Azure Disk), ou outros tipos.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: meu-pv-nfs
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany  # Pode ser montado por múltiplos nós em modo leitura/escrita
  persistentVolumeReclaimPolicy: Retain  # Retém os dados mesmo após o PVC ser deletado
  nfs:
    path: /data/meu-app
    server: nfs-server.exemplo.com
```

Criando um PersistentVolumeClaim (PVC)

Um **PersistentVolumeClaim (PVC)** é a solicitação de armazenamento feita por um Pod. O Kubernetes tenta encontrar um PV que corresponda aos requisitos do PVC.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: meu-pvc-app
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Uma vez que o PVC é criado, o Kubernetes o "binda" a um PV disponível que atenda aos critérios.

Usando o PVC em um Pod

Finalmente, um Pod pode usar o PVC para montar o armazenamento persistente:

```
apiVersion: v1
kind: Pod
metadata:
  name: minha-app-stateful
spec:
  containers:
  - name: minha-app-container
    image: minha-imagem-app-stateful:latest
    volumeMounts:
    - name: meu-volume-persistente
      mountPath: "/var/lib/minha-app/data"
  volumes:
  - name: meu-volume-persistente
    persistentVolumeClaim:
      claimName: meu-pvc-app
```

Resultado: Essa sequência garante que, mesmo que o Pod minha-app-stateful seja destruído e recriado em outro nó, ele sempre montará o mesmo volume persistente, e os dados em /var/lib/minha-app/data permanecerão intactos. Essa é a base para rodar bancos de dados, filas de mensagens e outras aplicações stateful de forma confiável no Kubernetes.

Provisionamento Dinâmico e StorageClasses

A gestão manual de PersistentVolumes pode ser trabalhosa em clusters grandes ou dinâmicos. É aí que entra o **provisionamento dinâmico** e os **StorageClasses**. Em vez de um administrador criar PVs manualmente para cada necessidade, um StorageClass define como o armazenamento deve ser provisionado. Quando um PVC é criado e especifica um StorageClass, o Kubernetes automaticamente provisiona um PV correspondente usando o provisionador definido no StorageClass.

O que é um StorageClass?

Um **StorageClass** é um objeto que descreve as "classes" de armazenamento disponíveis. Por exemplo, você pode ter um StorageClass para "armazenamento rápido" (usando SSDs) e outro para "armazenamento padrão" (usando HDDs). Cada StorageClass tem um provisionador (que sabe como criar o volume no backend de armazenamento) e parâmetros específicos.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-storage
provisioner: kubernetes.io/aws-ebs  # Exemplo para AWS EBS
parameters:
  type: gp2
  fsType: ext4
reclaimPolicy: Delete  # Deleta o PV e o volume físico quando o PVC é deletado
volumeBindingMode: Immediate
```

Usando um StorageClass em um PVC

Quando um PVC é criado, ele pode referenciar um StorageClass:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: meu-pvc-fast
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: fast-storage  # Referencia o StorageClass
```

Quando este PVC é criado, o Kubernetes usará o fast-storage StorageClass para instruir o provisionador (neste caso, o provisionador AWS EBS) a criar um volume EBS de 20Gi e, em seguida, criará um PV correspondente e o bindará ao PVC.

01

PVC Criado

Usuário cria um PVC especificando um StorageClass

02

Provisionador Acionado

Kubernetes aciona o provisionador definido no StorageClass

03

Volume Criado

Provisionador cria o volume físico no backend de armazenamento

04

PV Gerado

Kubernetes cria automaticamente um PV correspondente

05

Binding Completo

PV é bindado ao PVC e está pronto para uso pelo Pod

Resultado: Essa abordagem simplifica enormemente a gestão de armazenamento, permitindo que os desenvolvedores solicitem o tipo de armazenamento de que precisam, sem se preocupar com os detalhes de infraestrutura, e que os administradores definam políticas de armazenamento de forma centralizada. É um recurso essencial para a automação e escalabilidade de aplicações stateful em ambientes de nuvem e on-premise.

Consolidação dos Objetos Essenciais do Kubernetes

Chegamos ao fim de nossa jornada pelos principais objetos do Kubernetes, e agora você tem uma compreensão mais profunda de como ConfigMaps, Secrets, Ingress e PersistentVolumes/PersistentVolumeClaims trabalham juntos para construir e gerenciar aplicações robustas e escaláveis. Vimos que ConfigMaps são a chave para gerenciar configurações não sensíveis, permitindo flexibilidade e desacoplamento do código. Secrets, por sua vez, são o cofre para seus dados mais sensíveis, exigindo atenção especial à segurança e controle de acesso.

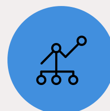
O Ingress emergiu como o roteador inteligente para expor seus serviços HTTP/HTTPS ao mundo externo, oferecendo roteamento baseado em host e caminho, além de terminação SSL/TLS. E, finalmente, PersistentVolumes e PersistentVolumeClaims são a espinha dorsal para garantir que suas aplicações stateful possam armazenar dados de forma duradoura, independentemente da efemeridade dos contêineres, com o auxílio dos StorageClasses para provisionamento dinâmico.

ConfigMaps

Gerenciamento de configurações não sensíveis com flexibilidade e desacoplamento

Secrets

Proteção de dados sensíveis com controle de acesso rigoroso e integração KMS



Ingress

Roteamento inteligente e terminação SSL/TLS para exposição de serviços HTTP/HTTPS

PV/PVC

Persistência de dados garantida com provisionamento dinâmico via StorageClasses

Em Prática

Ao projetar sua próxima aplicação no Kubernetes, comece pensando em como suas configurações serão externalizadas (ConfigMaps), como seus dados sensíveis serão protegidos (Secrets), como seus usuários acessarão a aplicação (Ingress) e como seus dados persistirão (PV/PVC). Integrar esses objetos desde o início garantirá uma arquitetura mais resiliente, segura e fácil de manter.

Autoavaliação

1

Diferença entre ConfigMap e Secret

Qual a principal diferença entre um ConfigMap e um Secret em termos de segurança?

- a) ConfigMaps são criptografados por padrão, enquanto Secrets são apenas codificados em Base64.
- b) Secrets são criptografados por padrão, enquanto ConfigMaps são armazenados em texto plano.
- c) ConfigMaps são para dados públicos, Secrets para dados sensíveis, ambos codificados em Base64 por padrão.
- d) ConfigMaps são para dados não sensíveis em texto plano, Secrets para dados sensíveis codificados em Base64, com segurança adicional via RBAC e KMS.

2

Exposição de Serviços

Um desenvolvedor precisa expor uma aplicação web (frontend) e uma API (backend) através do mesmo domínio minhaempresa.com, mas em caminhos diferentes (/ para o frontend e /api para o backend). Qual objeto do Kubernetes é mais adequado para essa tarefa?

- a) Service do tipo NodePort
- b) Service do tipo LoadBalancer
- c) Ingress
- d) ConfigMap

3

Persistência de Dados

Uma aplicação de banco de dados em um Pod Kubernetes precisa garantir que seus dados não sejam perdidos se o Pod for reiniciado ou movido para outro nó. Qual combinação de objetos do Kubernetes deve ser utilizada para resolver esse problema?

- a) ConfigMap e Secret
- b) Ingress e Service
- c) PersistentVolume e PersistentVolumeClaim
- d) Deployment e ReplicaSet

4

Ingress Controller

Qual das seguintes afirmações sobre o Ingress Controller é verdadeira?

- a) O Ingress Controller é um objeto API que define regras de roteamento.
- b) O Ingress Controller é um tipo de Service que expõe portas estáticas nos nós.
- c) O Ingress Controller é uma aplicação que monitora recursos Ingress e configura um proxy reverso.
- d) O Ingress Controller é responsável por provisionar armazenamento persistente para os Pods.

5

Provisionamento Dinâmico

Explique como o provisionamento dinâmico de PersistentVolumes, utilizando StorageClasses, simplifica a gestão de armazenamento em um cluster Kubernetes.

Gabarito

Questão 1

Resposta: d) ConfigMaps são para dados não sensíveis em texto plano, Secrets para dados sensíveis codificados em Base64, com segurança adicional via RBAC e KMS.

Questão 2

Resposta: c) Ingress

Questão 3

Resposta: c) PersistentVolume e PersistentVolumeClaim

Questão 4

Resposta: c) O Ingress Controller é uma aplicação que monitora recursos Ingress e configura um proxy reverso.

Próxima Aula

Aula 24

Padrões de Deploy no Kubernetes

Na **Aula 24 – Padrões de Deploy no Kubernetes**, exploraremos as estratégias mais eficazes para implantar e atualizar suas aplicações no Kubernetes, garantindo alta disponibilidade e minimizando o tempo de inatividade. Abordaremos padrões como Rolling Updates, Canary Deployments e Blue/Green Deployments, e como eles se integram com os objetos que aprendemos até agora.

Recursos Adicionais

📚 Documentação Oficial do Kubernetes

A fonte mais completa e atualizada para todos os objetos e conceitos.

📖 Livros e Cursos sobre Kubernetes Avançado

Para aprofundar-se em tópicos como segurança, monitoramento e otimização.

🌐 Blogs e Artigos da CNCF

Mantenha-se atualizado sobre as tendências e melhores práticas do ecossistema.

NOTA IMPORTANTE: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.