XPe

Aprenda com quem faz

Desenho de Arquiteturas de Dados Escaláveis

Mayla Taiane Teixeira

2022

SUMÁRIO

Capítulo 1.Arquitetura de Microsserviços

Escalabilidade

As aplicações em ambiente de produção costumam ter um uso dinâmico. É muito provável que um sistema de *ecommerce*, por exemplo, apresentará uma demanda muito grande em épocas comemorativas como no Natal e também na Black Friday. Já em dias comuns esse fluxo será bem menor.

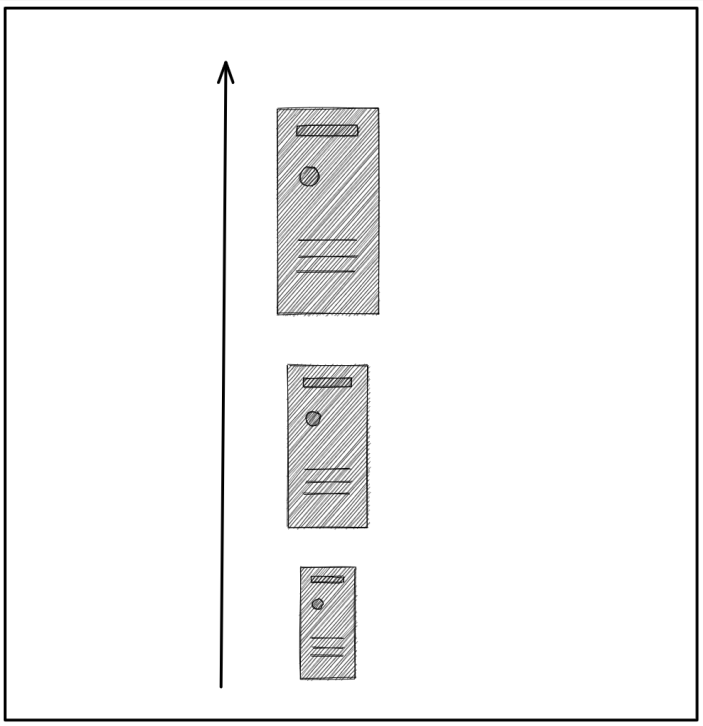
Diante disso, não é difícil perceber que a quantidade de recursos que o sistema precisa para garantir um funcionamento adequado pode apresentar variações ao longo do tempo. Deixar o seu sistema sempre com alta capacidade, baseando-se no maior consumo, representará um desperdício computacional/financeiro. O ideal é que esses recursos estejam sempre alocados de acordo com a demanda. Esse é o principal conceito de **escalabilidade**.

Escalabilidade Vertical

Escalar a sua aplicação verticalmente significa aumentar os recursos de hardware da máquina utilizada. Por exemplo: aumento do disco rígido, memória RAM ou CPU.

Neste caso, você pode esbarrar no limite de hardware disponível para uma única máquina. É importante se atentar também para o fato de que sua aplicação mantém-se em um único servidor e em caso de falha física, por exemplo, todo o sistema é afetado.

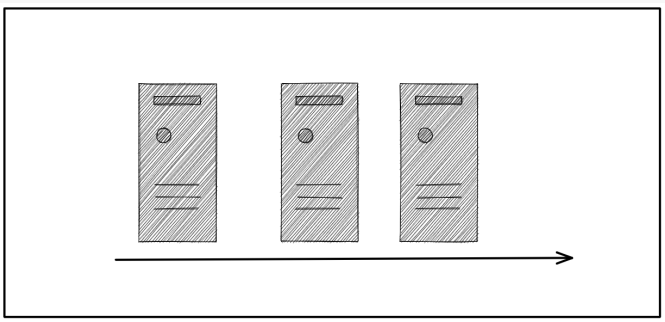
Figura 1 – Escalando Verticalmente.



Escalabilidade Horizontal

Escalar a sua aplicação horizontalmente significa aumentar a quantidade de réplicas dela. Dessa forma, é possível expandir a capacidade aumentando o número de máquinas. Cada máquina possui uma cópia do seu sistema e, caso ocorra falha em uma específica, as outras continuam funcionando normalmente.

Figura 2 – Escalando Horizontalmente.



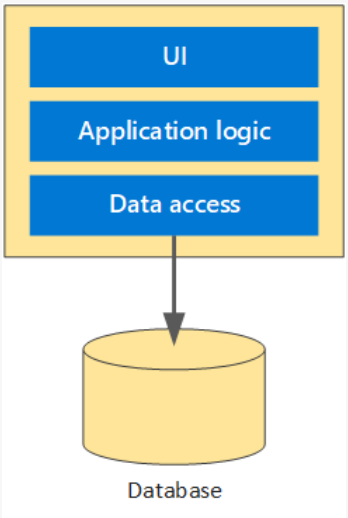
Microsserviços

Em tempo de desenvolvimento, normalmente as aplicações são construídas de forma incremental. A solução final é composta de módulos e, a cada funcionalidade nova, outros módulos podem ser adicionados.

No entanto, dependendo da arquitetura do sistema, em tempo de execução esses módulos são executados como um único processo. Ou seja, todos os componentes são empacotados gerando um artefato final, é o que chamamos de **Monolito.**

Na figura 3 abaixo, é possível verificar uma arquitetura típica de um monolito organizado em camadas de interface de usuário, lógica de aplicação e acesso a dados.

Figura 3 – Arquitetura de Monolítica



Fonte:
https://docs.microsoft.com/pt-br/azure/architecture/microservices/images/monolith/figure1.png

Essa arquitetura, à medida que vai crescendo e ganhando novas funcionalidades, tende a apresentar alguns problemas, como:

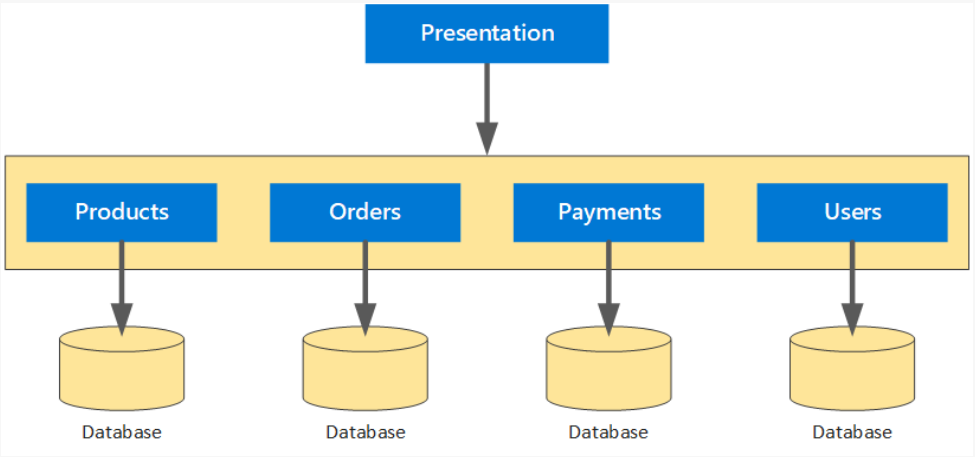
- O forte acoplamento entre os módulos pode fazer com que uma mudança específica cause impacto em vários outros pontos do

sistema. Demandando uma série de validações rigorosas a cada lançamento de versão.

- Os ciclos de releases se tornam mais lentos devido à quantidade de componentes e testes a serem realizados.
- dificuldade para escalar, uma vez que, mesmo que a demanda maior seja de um módulo específico, escalar horizontalmente necessita replicar toda a aplicação. A escalabilidade vertical pode ser aplicada, mas conhecemos suas limitações.

Visando resolver esses gargalos, algumas empresas passaram a migrar os seus monolitos para uma arquitetura de microsserviços. Nessa arquitetura os módulos do sistema são separados não apenas em tempo de desenvolvimento, mas também em tempo de execução (Valente, 2020).

Figura 4 – Arquitetura de Microsserviços.



Fonte: https://docs.microsoft.com/pt-br/azure/architecture/microservices/images/monolith/figure2.png

O fato de decompor o sistema em vários microsserviços traz algumas vantagens:

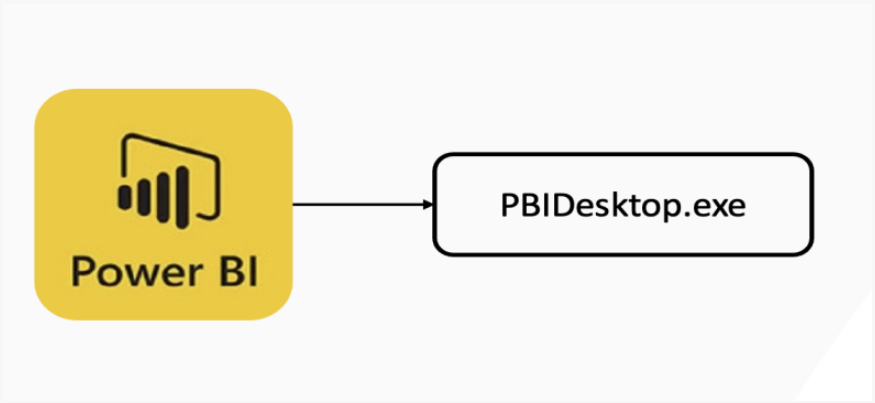
- Facilidade em lançar novas versões, uma vez que as unidades do sistema são menores e podem ser geradas separadamente.
- Facilidade de escalar, pois é possível detectar quais serviços estão com mais demanda e replicar somente eles. Simplificando o escalonamento horizontal.
- Permite falhas individuais: um serviço pode falhar e o sistema continuar funcionando de forma parcial.

Exemplo de Monolito

Para citar um exemplo de uma aplicação monolítica, podemos utilizar o PowerBI. O Power BI é uma das ferramentas mais conhecidas e mais utilizadas no mundo de dados para a visualização e análise de dados estratégica. Sem dúvida, é um líder de mercado.

A arquitetura da aplicação possui um desenho monolítico. Todos os componentes da aplicação estão contidos em um único artefato nomeado PBIDesktop.exe, que fica disponível em nossa máquina de trabalho (Fig. 4).

Figura 4 – Power BI.



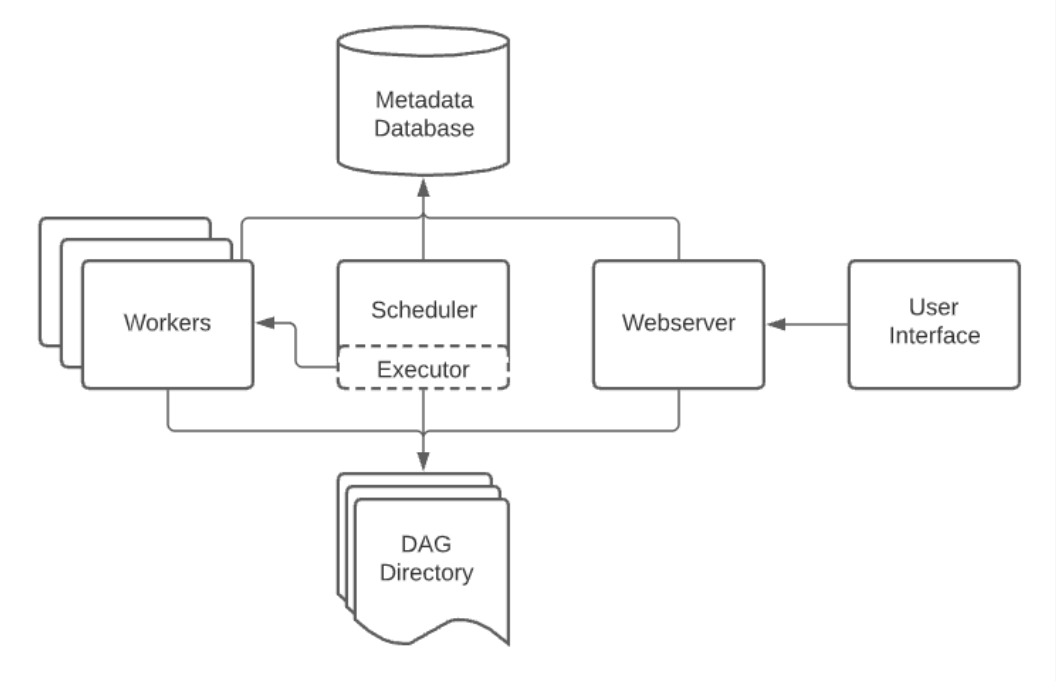
Perceba que para atualizar um único componente do Power BI, todo o executável PBIDestktop.exe deve ser atualizado e novamente instalado na

máquina. Da mesma forma, se algum componente do PowerBI apresentar falha, todo o sistema deve ser atualizado e reinstalado no servidor.

Exemplo de Microsserviço

Um exemplo de aplicação baseada numa arquitetura de microsserviços é o Apache Airflow. Essa ferramenta é bastante conhecida no ambiente de Engenharia de Dados, utilizada para agendamento e monitoramento de pipeline de dados.

Figura 5 – Arquitetura do Apache Airflow.



Fonte: https://airflow.apache.org/docs/apache-airflow/stable/_images/arch-diag-basic.png

A figura 5 ilustra a arquitetura do Apache Airflow, abaixo uma descrição desses componentes:

- Database de metadados, responsável pelo armazenamento dos dados da aplicação.

- O Webeserver, com o qual o usuário vai interagir para o controle de DAGs (pipelines do Airflow).
- O Scheduler, responsável pela organização e distribuição das execuções das tarefas no Airflow.
- Os Workers, instâncias nas quais a computação das tarefas acontece de fato.
- O Diretório de DAGs, uma pasta de sistema, volume ou repositório git em que os códigos das DAGs são armazenados.

No gerenciamento de pipelines, há alguns cenários possíveis relacionados ao escalonamento de recursos. No caso de haver poucos usuários do webserver e uma carga grande de processamento relacionada às tasks, é possível escalar apenas os workers de execução para dar, a cada tarefa, os recursos necessários. No caso de haver tasks com pouca necessidade de poder de processamento e muitos usuários simultâneos, podemos escalar apenas o webserver. No caso de haver DAGs com um alto grau de complexidade e paralelização, por exemplo, é possível escalar o scheduler. A alocação de recursos para cada um desses componentes é totalmente modificável de maneira dinâmica de acordo com o tipo de consumo e funcionamento dos pipelines.

> Capítulo 2

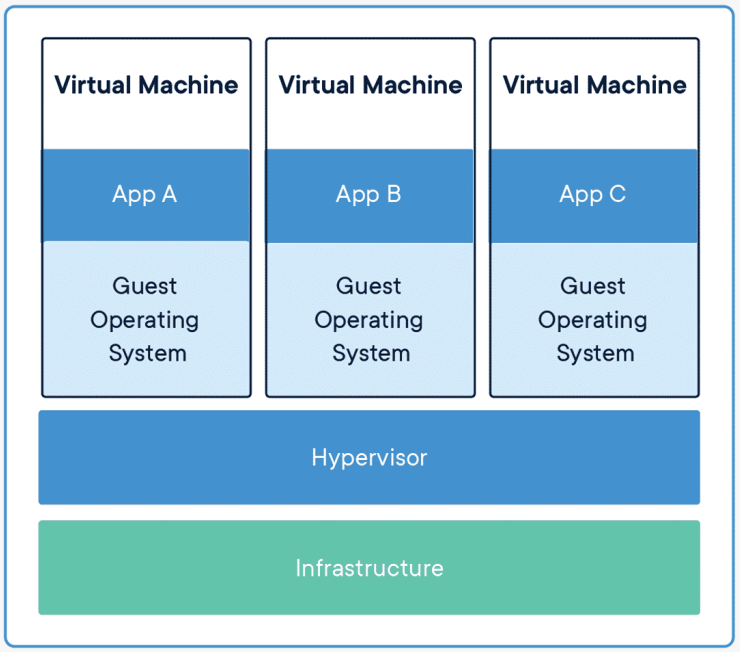
Capítulo 2.Docker

Máquinas Virtuais e Containers

Uma necessidade muito comum na área de TI é a replicação de ambientes. Seja para situações de desenvolvimento, como garantir que todos os desenvolvedores tenham o mesmo cenário, ou em produção, para que as aplicações estejam empacotadas e com todas as dependências necessárias para o seu funcionamento. Esse processo pode ser resolvido por meio da utilização de máquinas virtuais ou containers.

Nas máquinas virtuais (VM) ocorre a virtualização de todos os componentes necessários, incluindo hardware e o sistema operacional por completo. Além disso, as máquinas virtuais, por simularem servidores por completo, são mais pesadas. A figura 6 ilustra a estrutura para execução das VMs.

Figura 6 –Máquinas Virtuais



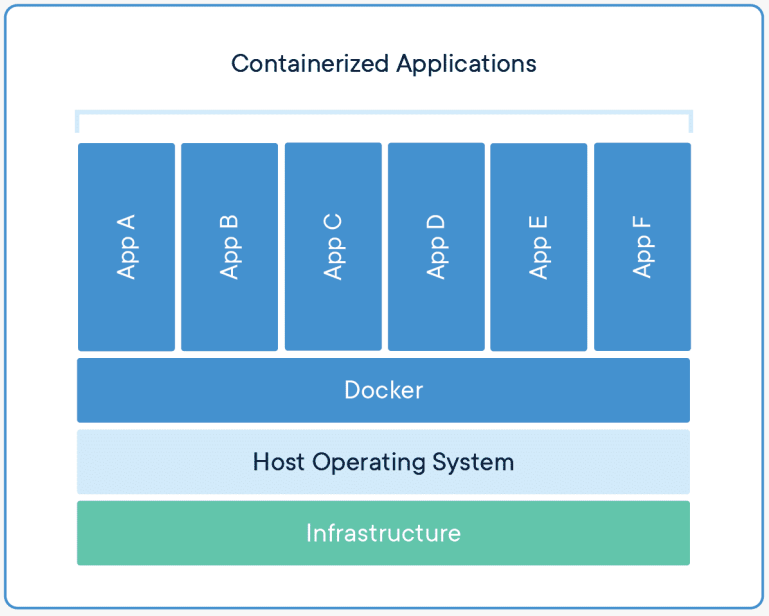
Fonte: https://www.docker.com/wp-content/uploads/2021/11/container-vm-whatcontainer_2.png

Containers são bem mais leves que as VMs, eles empacotam somente os recursos necessários para executar a aplicação e compartilham o Kernel do sistema operacional com outros containers.

Cada container roda naturalmente de forma isolada. No entanto, algumas otimizações são feitas para aumentar a eficiência. Como eles são construídos em camadas, o runtime do container realiza um sistema de cache, fazendo o download apenas daquilo que é necessário, otimizando o espaço em disco e o uso de largura de banda de rede (ARUNDEL e DOMINGUS, 2019).

A partir de um arquivo contendo um conjunto de instruções necessárias, empacotamos a nossa aplicação no formato de imagem. Ao executar essa imagem em um runtime, um processo é criado e a ele atribuímos o nome de container. A figura 7 ilustra a estrutura para execução dos containers, no caso ilustrado o Docker é considerado como o CRI (Container Runtime Interface).

Figura 7 –Containers



Fonte: https://www.docker.com/wp-content/uploads/2021/11/container-vm-whatcontainer_2.png

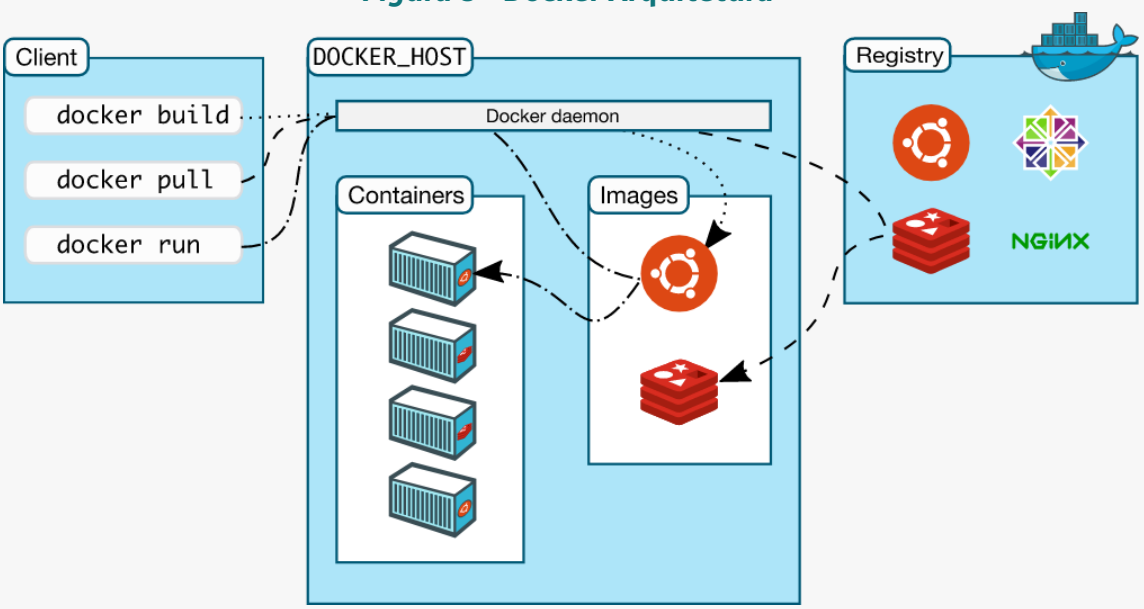
Veja mais sobre o comparativo entre containers e Vms em: https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

Docker: principais conceitos

O **Docker** se refere a um conjunto de soluções que auxiliam na construção, execução e gerenciamento de containers (https://docs.docker.com/get-started/overview/).

Em uma arquitetura cliente-servidor, como mostrada na figura 8, o Docker *client* comunica com o Docker *daemon* através de uma REST API. O Docker *client* envia comandos para o Docker *daemon*, que é responsável por executar as ações de criação, manipulação e distribuição dos containers.

Figura 8 –Docker Arquitetura



Fonte: https://docs.docker.com/engine/images/architecture.svg

O componente Registry da figura 8 é o responsável pelo armazenamento das imagens. Trata-se de um registro remoto que contém diversas imagens identificadas por um nome único. O registro de container padrão do Docker é o Docker Hub que permite a criação de repositórios públicos e privados. O Docker Hub armazena mais de 100.000 imagens, dentre elas ferramentas open-source bastante utilizadas, como o NGINX, MongoDB, Redis, Apache Airflow entre outras (*https://hub.docker.com/*).

Instalando o Docker

Para instalação, siga as instruções de acordo com o seu sistema operacional https://docs.docker.com/get-docker/.

Docker: comandos básicos

Após a instalação do Docker poderemos ter acesso à ferramenta de linha de comando (CLI). É através dela que executaremos os comandos para criação, manipulação e remoção de containers.

Veja uma lista dos comandos básicos em: https://docs.docker.com/engine/reference/commandline/docker/.

Arquivo de definição de imagem

O Dockerfile é o código que define as camadas de composição de uma imagem de container. A figura 9 abaixo possui uma ilustração de um Dockerfile simples para build de uma imagem com um processo Python.

Figura 9 –Exemplo de um Dockerfile

```
1   FROM alpine
2
3   RUN apk add --no-cache python3 py3-pip
4
5   RUN pip3 install fastapi uvicorn
6
7   EXPOSE 80
8
9   COPY ./app /app
10
11  CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

No exemplo da figura 9, a primeira linha faz a importação de uma imagem do sistema operacional alpine, uma distribuição Linux bastante enxuta para aplicação com configurações mínimas. A linha 3 faz a instalação do python e do package manager pip. A linha 5 faz a instalação de duas bibliotecas python: fastapi e uvicorn. A linha 7 abre porta 80 do container para receber requisições. A linha 9 copia todo o conteúdo da pasta local ./app para a pasta /app dentro do container e a linha 11 declara o comando que será executado sempre que o container for iniciado.

> Capítulo 3

Capítulo 3.Kubernetes: introdução e instalação

Kubernetes

O processo de deploy e gerenciamento de aplicações no ambiente produtivo envolve inúmeras tarefas. Como visto anteriormente, o fato de conseguirmos conteinerizar nossas aplicações já facilita bastante esse processo. No entanto, tarefas como as de escalar, monitorar e reiniciar em caso de falhas vão se mostrando cada vez mais difíceis à medida que o número de aplicações aumenta.

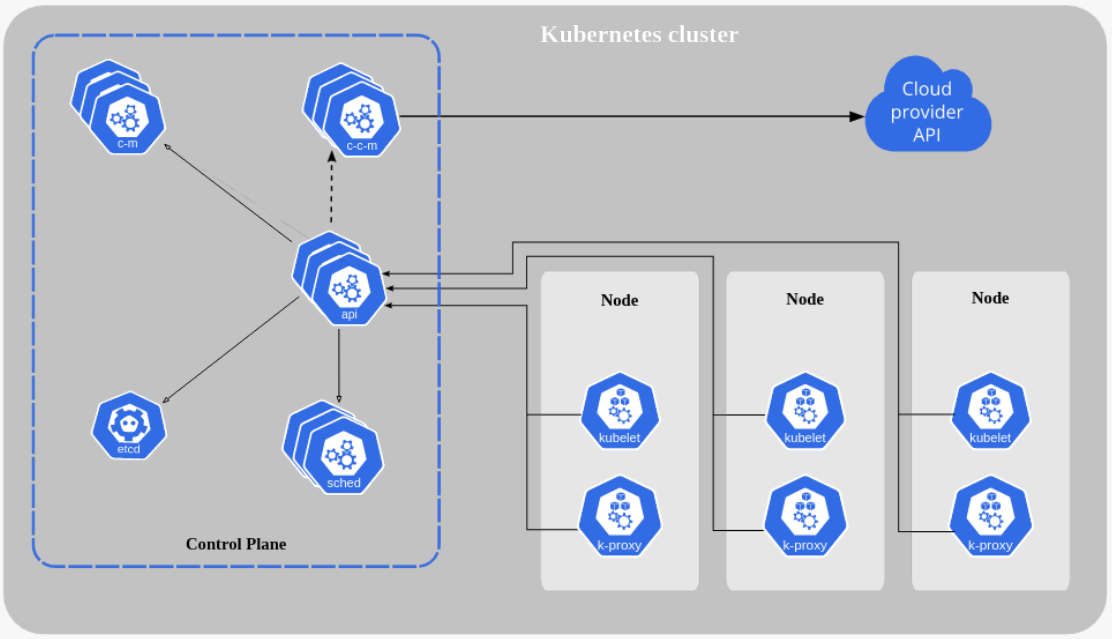
Nesse sentido, em 2014 o Google abriu para a comunidade um projeto interno, visando desenvolver um orquestrador de containers para resolver o problema de executar um grande número de serviços em escala, chamado Kubernetes. Como um produto gratuito e de código aberto a adoção foi muito grande e no ano de 2016 foi aceito na Cloud Native Computing Foundation (CNFC).

O Kubernetes, abreviado por k8s por conta das oito letras entre o “k” e o “s”, é uma plataforma de orquestração de containers, que oferece um ambiente escalável e confiável para o deploy e gerenciamento de aplicações.

Arquitetura do Kubernetes

Podemos dividir a arquitetura do K8s em dois componentes: o Plano de Controle (Control Plane) e Nós (Nodes). A figura 10 ilustra essa arquitetura.

Figura 10 –Kubernetes Arquitetura



Fonte: adaptado de
https://d33wubrfki0l68.cloudfront.net/2475489eaf20163ec0f54ddc1d92aa8d4c87c96b/e7c81/images/docs/components-of-kubernetes.svg

O Plano de controle é o cérebro do kubernetes, nele contém diversos componentes que dividem responsabilidades como as de escalonar containers, atender requisições de API, gerenciar serviços e etc.

Abaixo uma descrição de cada um desses componentes:

- **api**: o kube-apiserver é responsável por tratar as requisições HTTP, funcionando como uma interface para comunicação com o plano de controle.
- **etcd**: banco de dados que armazena todas as informações referentes ao funcionamento do cluster.
- **sched**: kube-scheduler é responsável por definir em qual nó um Pod recém-criado irá ser executado.
- **c-m**: kube-controller-manager contém os controladores de recursos do k8s.

- **c-c-m**: o cloud-controller-manager é um componente presente no caso de soluções em nuvem, ele é responsável por realizar a interface com o provedor de nuvem para criação de recursos como volumes de disco e balanceadores de carga.

Cada nó do cluster k8s é composto pelos seguintes componentes:

- **kubelet**: realiza o gerenciamento do nó e se comunica com o plano de controle.
- **k-proxy**: o kube-proxy é responsável pelo gerenciamento dos recursos de rede.
- container runtime: engine responsável pela execução de containers.

Veja mais detalhes sobre os componentes do Kubernetes na documentação oficial:

https://kubernetes.io/docs/concepts/overview/components/.

Instalando o Minikube

O Minikube é uma solução local para o Kubernetes, que tem por objetivo fornecer um ambiente de aprendizado e desenvolvimento. Para instalação, siga as instruções de acordo com o seu sistema operacional https://minikube.sigs.k8s.io/docs/start/.

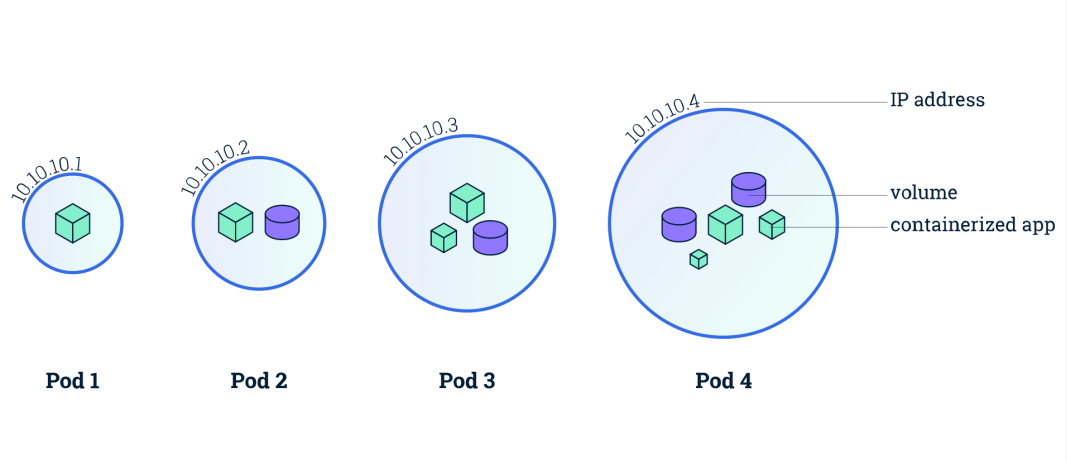
> Capítulo 4

Capítulo 4.Kubernetes: criando, expondo e escalando aplicações

O que é um Pod?

Pod é um objeto do Kubernetes que contém um ou mais containers que compartilham recursos de rede e de armazenamento.

Figura 11 –Pod



Fonte: https://d33wubrfki0l68.cloudfront.net/fe03f68d8ede9815184852ca2a4fd30325e5d15a/98064/docs/tutorials/kubernetes-basics/public/images/module_03_pods.svg

Ao implantar uma aplicação no Kubernetes, ela deve estar conteinerizada e este container será executado dentro de um Pod. Abaixo um exemplo de manifesto com especificação de um Pod:

```
apiVersion: v1

kind: Pod

metadata:

  name: nginx
```

```
    labels:
      app.kubernetes.io/name: MyApp
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
      - containerPort: 80
        name: http-web-svc
```

No exemplo acima estamos configurando um Pod com nome “nginx” que contém o container do nginx na versão 1.14.2. Além disso, estamos expondo a porta 80 desse container. Para aplicar essa configuração no cluster, salve esse conteúdo em um arquivo chamado nginx-pod.yaml e execute o comando abaixo:

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

Um Pod com o nome nginx será criado no namespace default. Ao listar, poderemos verificar esse objeto em execução:

```
kubectl get pods
```

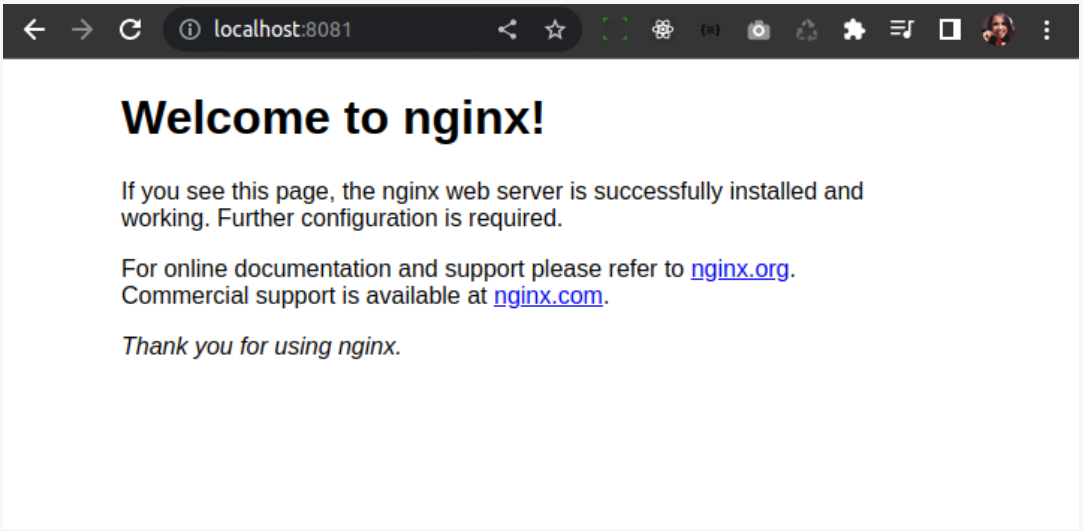
Para acessar a aplicação, podemos fazer o redirecionamento de porta:

```
kubectl port-forward pod/nginx 8081:80
```

Através do comando acima, estamos direcionando a porta 8081 da nossa máquina local para a porta 80 do container. Dessa forma, podemos

acessar o servidor nginx na url http://localhos:8081. Você deverá ver a seguinte tela:

Figura 12 –Servidor NGINX



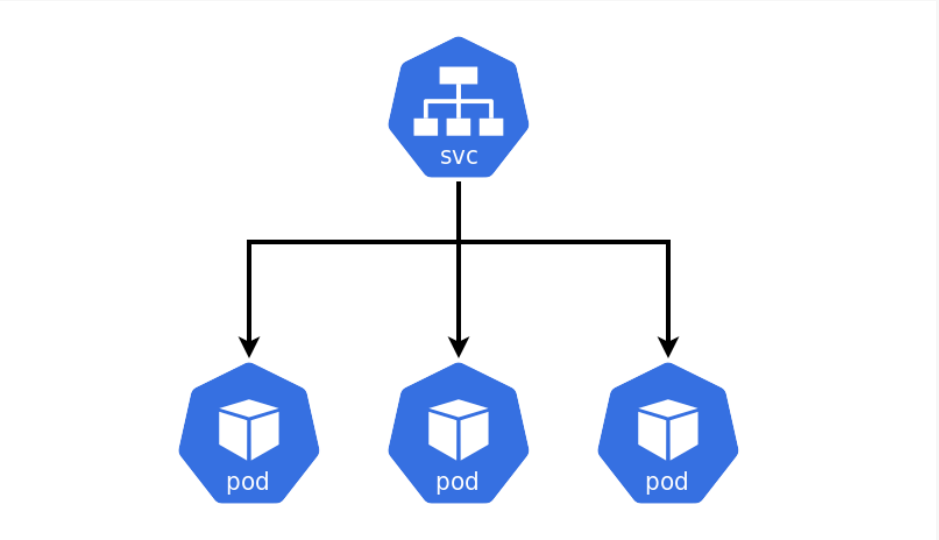
Para entender mais sobre Pods, acesse: https://kubernetes.io/docs/concepts/workloads/pods/

O que é um Service?

Containers dentro de um mesmo Pod podem se comunicar via *localhost*. No entanto, em muitos casos você vai precisar que ocorra comunicação entre Pods diferentes. Para viabilizar essa comunicação interna (entre Pods) ou até mesmo externa (acesso de fora do cluster) existem os Services. Um Service realiza a interface para acesso a uma unidade ou a um determinado conjunto de Pods.

Além de fornecer um IP único e imutável, um Service funciona como um balanceador de carga, por exemplo, para uma aplicação com várias réplicas. Na figura 13, temos um serviço fornecendo um caminho único para acesso a um determinado grupo de Pods.

Figura 13 –Service



Fonte: https://miro.medium.com/max/1400/1*evgmvxfGmkyOj06inJCvKw.png

Como exemplo, vamos criar um serviço para expor a aplicação que criamos anteriormente com o container do NGINX:

```
apiVersion: v1

kind: Service

metadata:

  name: nginx-service
```

```
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: http-web-svc
```

O serviço especificado acima, com nome de nginx-service faz o redirecionamento da comunicação TCP que chega na porta 80 para a porta *http-web-svc* de qualquer Pod que tenha a label *app.kubernetes.io/name: MyApp*. Para aplicar essa configuração no cluster, salve esse conteúdo em um arquivo chamado nginx-svc.yaml e execute o comando abaixo:

```
kubectl apply -f nginx-svc.yaml
```

Um Service com o nome nginx-service será criado no namespace default. Ao listar, poderemos verificar esse objeto em execução:

```
kubectl get svc
```

Certifique que o Pod criado no tópico anterior esteja em funcionamento e, dessa forma, será possível acessar a aplicação realizando o redirecionamento de porta do serviço criado:

```
kubectl port-forward svc/nginx-service 8082:80
```

Para acessar a aplicação, agora através do serviço, acesse a URL http://localhost:8082.

Para mais detalhes sobre os Services veja: https://kubernetes.io/docs/concepts/services-networking/service/

O que são Secrets e ConfigMaps?

Secrets e ConfigMaps são objetos utilizados para o armazenamento de um pequeno volume de dados para fins de configuração, no formato chave-valor dentro do Kubernetes. Eles podem posteriormente ser utilizados para atribuição de valores de variáveis de ambiente ou criação de volumes nos containers. A principal diferença entre os dois é que as Secrets foram construídas para o armazenamento de dados sensíveis, como senhas, chaves e tokens.

Para mais detalhes sobre configurações dentro do Kubernetes veja: https://kubernetes.io/docs/concepts/configuration/.

O que são ReplicaSets?

Um ReplicaSet é responsável por replicar e garantir a disponibilidade de Pods. Suponha que queiramos escalar a aplicação do NGINX que criamos anteriormente. Poderíamos criar um ReplicaSet especificando a quantidade de Pods que queremos manter disponíveis e ele cuidará de criar, monitorar e garantir que essa quantidade seja atendida e que os Pods estejam em execução.

```
apiVersion: apps/v1

kind: ReplicaSet

metadata:
```

```
  name: nginx-rs
  labels:
    app.kubernetes.io/name: MyApp
spec:
  replicas: 3
  selector:
    matchLabels:
      app.kubernetes.io/name: MyApp
  template:
    metadata:
      labels:
        app.kubernetes.io/name: MyApp
    spec:
      containers:
       - name: nginx
          image: nginx:1.14.2
          ports:
          - containerPort: 80
            name: http-web-svc
```

No exemplo acima, estamos solicitando 3 réplicas do Pod com as mesmas configurações anteriores. Após aplicar esse manifesto, execute o comando abaixo para listar os ReplicaSets:

```
kubectl get rs
```

Você pode listar os Pods e acompanhar as alterações feitas pelo ReplicaSet. Altere a quantidade de réplicas e veja isso acontecendo na prática.

```
kubectl get pods -w
```

Para mais detalhes sobre os ReplicaSets veja: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

> Capítulo 5

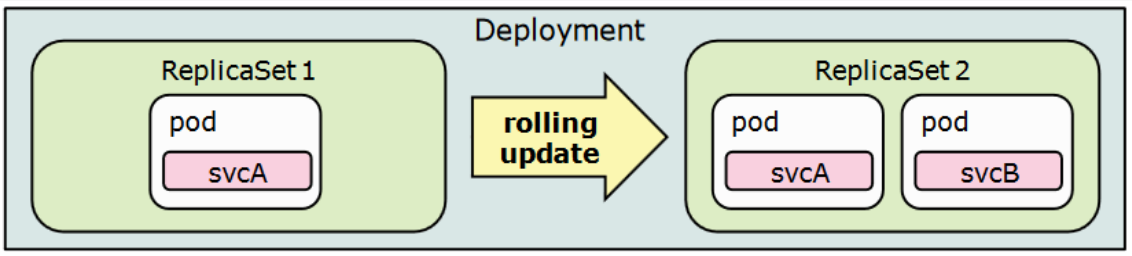
Capítulo 5.Kubernetes : gerenciando aplicações e persistindo dados

Deployments

Deployments são objetos que acrescentam uma camada a mais aos ReplicaSets. Ele tem, portanto, além de um ReplicaSet, todo um controle de versionamento de sua aplicação. Por meio dele, é possível acompanhar todas as versões aplicadas e inclusive voltar em uma delas, caso seja necessário. Uma recomendação presente na documentação do Kubernetes é que você use Deployment ao invés de ReplicaSet diretamente.

Na figura 14 temos ilustrado como o Deployment forma uma camada acima do ReplicaSet e controla a atualização do estado.

Figura 14 –Deployment



Fonte: https://www.ibm.com/cloud/architecture/images/courses/kubernetes-101/deployments-replica-sets-and-pods2.png

Para mais detalhes sobre os Deployments veja: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

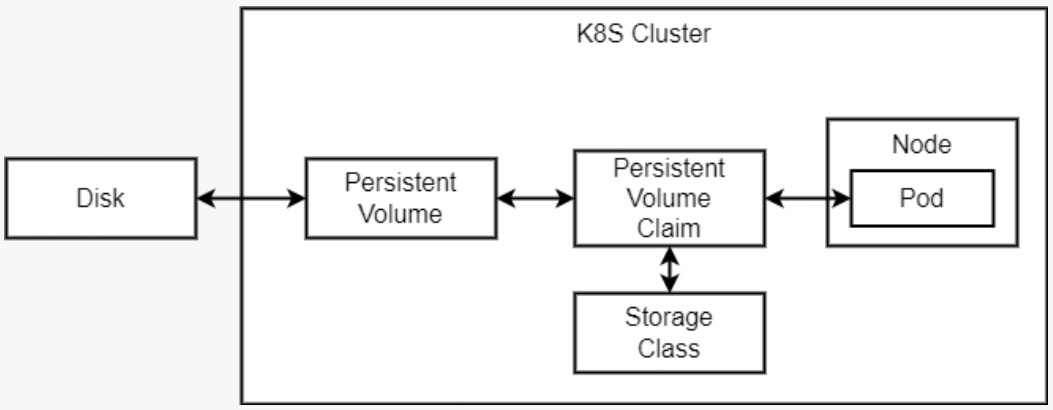
Armazenamento Persistente

Volumes são objetos Kubernetes para persistência de dados e compartilhamento de arquivos entre containers. Os volumes podem ser do tipo efêmero, que acompanha o ciclo de vida do Pod, ou seja, ele existe somente enquanto o Pod está em execução. Caso ocorra alguma falha, ou o Pod finalize, o volume também é removido. Para aplicações mais avançadas, que precisam manter estados caso sejam reiniciadas, existem os volumes persistentes (PersistentVolume).

Um PersistentVolume (PV) é responsável por fornecer uma forma de armazenamento persistente, abstraindo toda a forma como esse processo é feito nos diversos provedores. Para criar PVs de forma dinâmica são utilizadas as StorageClass.

Um determinado Pod acessa um PersistentVolume por meio de um PersistentVolumeClaim (PVC). Este último é responsável por realizar a requisição dos recursos de espaço e modo de acesso a um determinado PV.

Figura 15 –Persistência no Kubernetes



Fonte: https://blog.dkwr.de/k8s/dynamicdisk/persistentvolumeclaim.drawio.png

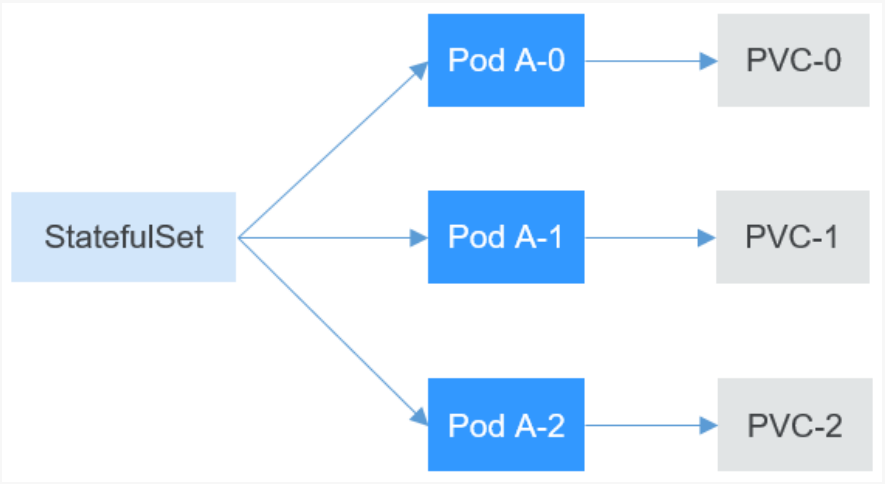
Para mais detalhes sobre armazenamento de dados no Kubernetes veja: https://kubernetes.io/docs/concepts/storage/

StatefulSets

StatefulSets são objetos que fornecem, além das funcionalidades de um Deployment, a garantia de um identificador único para cada Pod e a ordenação deles. Essa garantia é importante para aplicações stateful, ou seja, aplicações que necessitam do armazenamento de estado.

Vamos tomar como exemplo uma aplicação distribuída de banco de dados como o Redis ou o Cassandra. Neste caso, os Pods que representam os nós do cluster precisam manter a identificação e o mesmo volume do Pod anterior em caso de recriação. Os StatefulSets se apresentam como solução ideal para esses casos. Veja na figura 16 uma pequena ilustração do componente StatefulSet com três pods ordenados.

Figura 16 – StatefulSets



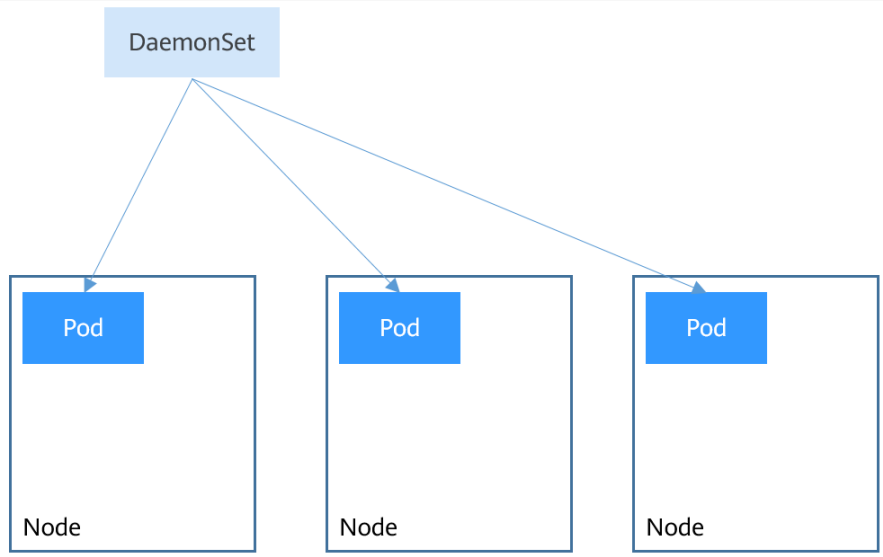
Fonte: https://support.huaweicloud.com/intl/en-us/basics-cce/en-us_image_0258203193.png

Para mais detalhes sobre o StatefulSet veja:
https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

DaemonSet

DaemonSet é um objeto que garante uma réplica do Pod em cada nó do cluster Kubernentes. Esse recurso pode ser utilizado, por exemplo, em caso de aplicações de coleta de logs ou de monitoramento. Na figura 17 é possível visualizar um exemplo de um cluster kubernentes com 3 nós e um recurso DaemonSet criando um Pod em cada um destes nós.

Figura 17 – DaemonSet



Fonte: https://support.huaweicloud.com/intl/en-us/eu-west-0-usermanual-cce/en-us_image_0000001243779385.png

Para mais detalhes sobre o DaemonSet veja:
https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

Jobs e CronJobs

Um Job é responsável por criar e controlar um ou mais Pods até que um determinado número deles sejam executados com sucesso. Esse recurso é ideal para tarefas em lote (batch), que têm um tempo de vida limitado.

É possível criar agendamento de execução de um recurso Job, para isso o kubernentes oferece o recurso CronJob.

Veja mais detalhes sobre esses recursos na documentação do kubernetes: https://kubernetes.io/docs/concepts/workloads/controllers/job/ e https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/

> Capítulo 6

Capítulo 6.Helm Charts

Introdução

Helm é um gerenciador de pacotes para Kubernetes. Esses pacotes são gerados no formato de *charts*. Um chart possui um conjunto de arquivos que contém os parâmetros de configuração, bem como todas as definições dos recursos necessários para execução da sua aplicação (https://helm.sh/docs/topics/charts/).

Os Helm Charts também possuem repositórios remotos onde os charts são armazenados e disponibilizados. Um repositório remoto pode ser qualquer servidor HTTP que serve arquivos YAML, para isso podem ser utilizados um bucket S3 ou GCS (Google Cloud Storage), Github Pages ou qualquer servidor Web. O Artifact Hub (https://artifacthub.io/) é o repositório padrão do Helm. Bastante utilizado pela comunidade, nele é possível localizar charts oficiais como do Prometheus e do Grafana que veremos no capítulo 8.

Instalando o Helm CLI

A instalação da ferramenta de linha de comando para utilização do Helm pode ser feita através do gerenciador de pacotes do seu sistema operacional.

Instalação no Linux

```
curl https://baltocdn.com/helm/signing.asc | gpg --dearmor \

| sudo tee /usr/share/keyrings/helm.gpg > /dev/null

sudo apt-get install apt-transport-https --yes

echo "deb [arch=$(dpkg --print-architecture) \
```

signed-by=/usr/share/keyrings/helm.gpg] \

https://baltocdn.com/helm/stable/debian/ all main" | \

sudo tee /etc/apt/sources.list.d/helm-stable-debian.list

sudo apt-get update

sudo apt-get install helm

- Instalação no Windows

  choco install kubernetes-helm

- Outras opções de instalação podem ser encontradas no site oficial do Helm: https://helm.sh/docs/intro/install/.

Valide a instalação do Helm CLI, executando o comando abaixo:

helm version

Se a instalação foi feita com sucesso, você terá uma saída similar a essa:

version.BuildInfo{

Version:"v3.7.1",

GitCommit:"1d11fcb5d3f3bf00dbe6fe31b8412839a96b3dc4",

GitTreeState:"clean",

GoVersion:"go1.16.9"}

Instalando e gerenciando um Helm Chart

Com a ferramenta de linha de comando instalada, é possível realizar a instalação de qualquer Helm Chart de forma simples e prática.

OBS: aqui estará um breve tutorial conforme a aula de deploy do Airflow

> Capítulo 7

Capítulo 7.Operators

Introdução

Como já vimos nos capítulos anteriores, o Kubernetes e seus recursos nativos conseguem automatizar inúmeras tarefas relativas à implantação e gerenciamento de aplicações. No entanto, podem existir tarefas peculiares de seu sistema, como realização de um backup regular ou de um deploy específico conforme demanda, que você provavelmente gostaria que também fossem automatizadas.

Neste sentido, um dos pontos excepcionais do Kubernetes é fornecer uma API totalmente extensível. Essa funcionalidade pode ser explorada construindo recursos customizados, que tem por objetivo automatizar ao máximo os processos de sua aplicação.

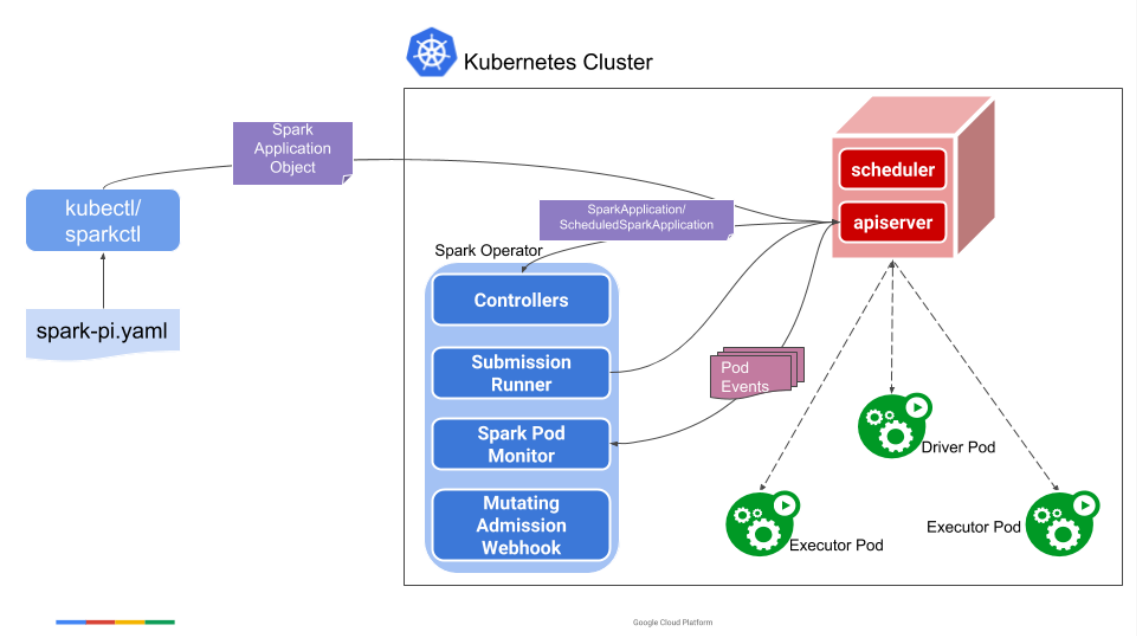
Os softwares capazes de estender as funcionalidades do Kubernetes, criando recursos customizados para o gerenciamento de aplicações especificas, são chamados de Operators (https://kubernetes.io/docs/concepts/extend-kubernetes/operator/).

Spark Operator

Um operador muito conhecido e de grande utilidade na Engenharia de Dados é o Spark Operator. Um projeto de código aberto, que permite rodar aplicações Spark no Kubernetes de maneira extremamente simples.

A figura 18 ilustra os diferentes recursos que compõem a arquitetura do Spark Operator. Na prática a aplicação Spark é especificada por meio de um arquivo YAML, ao aplicar esse manifesto no Kubernetes, um objeto chamado *SparkApplication* é criado e os componentes do Operator são responsáveis por criar, gerenciar e monitorar os recursos do cluster Spark.

Figura 18 –Arquitetura do Spark Operator



Fonte: https://raw.githubusercontent.com/GoogleCloudPlatform/spark-on-k8s-operator/v1beta2-1.3.7-3.1.1/docs/architecture-diagram.png

A documentação completa sobre o funcionamento deste operador e a especificação das *SparkApplication*, pode ser encontrada no repositório oficial: https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/v1beta2-1.3.7-3.1.1/docs/quick-start-guide.md

Instalando o Spark Operator

Iremos utilizar o Helm Chart do Spark Operator para fazer a instalação. Para isso adicione o repositório remoto executando o comando abaixo:

```
helm repo add spark-operator \
```

https://googlecloudplatform.github.io/spark-on-k8s-operator

E em seguida realize a instalação do chart:

```
helm install spark-operator spark-operator/spark-operator \
--namespace processing --create-namespace
```

Verifique se o operador está em execução

```
helm status --namespace processing spark-operator
```

Agora que temos nosso operador em execução, podemos fazer o deploy de uma aplicação Spark. Para isso, vamos pegar um código exemplo, disponível no repositório oficial (https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/v1beta2-1.3.7-3.1.1/examples/spark-py-pi.yaml). Copie o código desse exemplo em um arquivo local chamado spark-py-pi.yaml e realize o apply no cluster.

```
kubectl apply -f spark-py-pi.yaml -n processing
```

Podemos visualizar os eventos da aplicação criada executando o comando abaixo:

```
kubectl describe sparkapplication spark-pi -n processing
```

> Capítulo 8

Capítulo 8.Monitoramento

Métricas

Métricas são medidas numéricas que descrevem o uso ou o comportamento de sistemas. Em uma Aplicação Web, por exemplo, você pode ter métricas como: número de requisições, tempo de resposta da requisição e quantidade de erros. Já em um banco de dados podemos ter quantidade de conexões ativas, uso de CPU, uso de espaço em disco e outros.

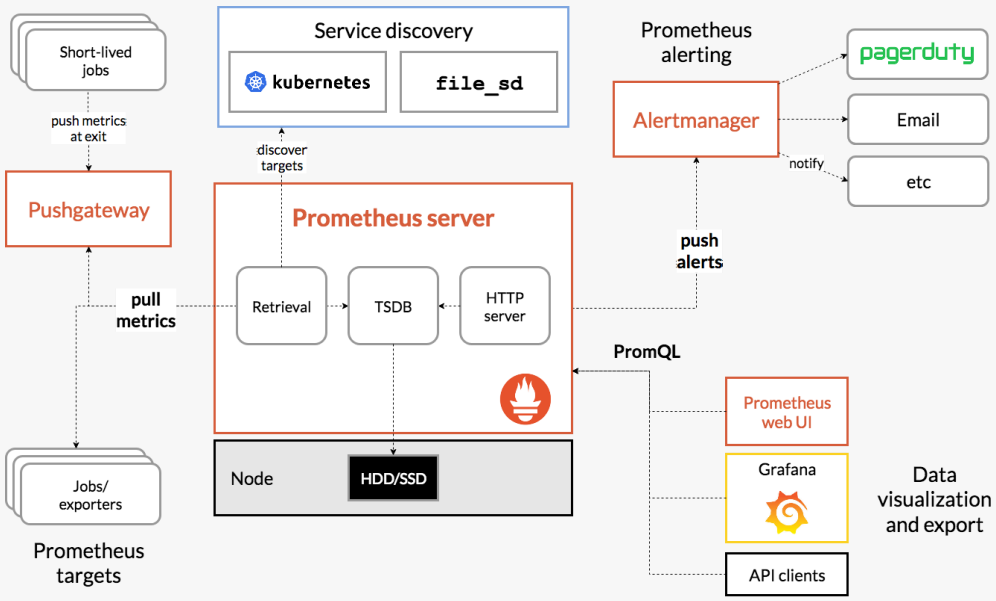
Métricas são muito importantes para a gestão das aplicações em um ambiente de produção. Através delas você pode ter *insights* sobre o uso e tomar decisões como aumentar os recursos disponíveis, separar um determinado módulo em um microsserviço, reduzir quantidade de máquinas, etc.

Prometheus

O Prometheus (https://prometheus.io/) é uma ferramenta de código aberto criada pela SoundCloud em 2012 e integrada ao Cloud Native Computing Foundation (CNCF) em 2016. Com ela é possível coletar métricas de aplicações e criar alertas conforme necessidade.

Na figura 19 é possível visualizar os diferentes componentes dessa ferramenta. O Prometheus Server é o coração dessa arquitetura. Ele é responsável por realizar a coleta das métricas através do *Retrieval*, armazená-las em um banco de dados de série temporal (TSDB) e servi-las em uma API pelo HTTP Server.

Figura 19 –Arquitetura do Prometheus



Fonte: https://prometheus.io/assets/architecture.png

O *Service Discovery* é uma funcionalidade extremamente útil. Por meio dela, o Prometheus, assim que implantado, realiza a identificação das aplicações que estão servindo as métricas e inicia a coleta delas.

O Prometheus realiza a coleta das métricas de forma ativa. Ou seja, ele vai até as aplicações e recupera as métricas disponíveis. Chamamos esse mecanismo de *pull metrics* e a frequência em que ele ocorre é configurável. Em caso de aplicações com tempo de vida curto, pode ser que não dê tempo da coleta ser realizada. Para esses casos, a aplicação deve enviar as métricas ao componente *Pushgateway* que então, irá servi-las ao Prometheus Server.

As aplicações podem gerar métricas de forma nativa. Para isso, existem várias bibliotecas disponíveis nas diversas linguagens como Python, Go, Java e Scala. Para casos em que as métricas não são exportadas nativamente, componentes chamados *Exporters* podem ser

implementados. Eles serão responsáveis por gerar e disponibilizar essas métricas.

As métricas coletadas pelo Prometheus podem ser consultadas utilizando uma linguagem própria, chamada PromQL. Por meio dessa interface é possível integrar outras ferramentas como o Grafana para criação de Dashboards.

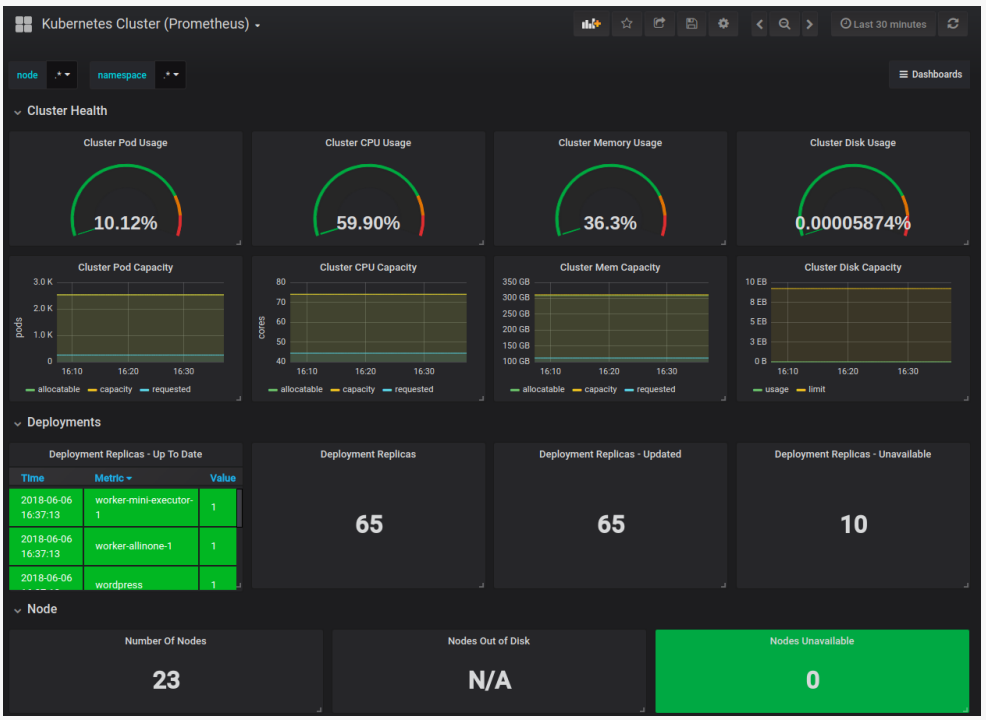
Uma funcionalidade importante, também disponível no Prometheus, é a geração de alertas. Através do *Alertmanager* é possível combinar diversas métricas e gerar alertas que podem ser disparados via e-mail, slack e outros.

Grafana

Grafana OSS (Open Souce Software) é uma ferramenta que permite a realização de consultas, criação de visualizações e geração de alertas para métricas, logs e traces previamente armazenadas (https://grafana.com/oss/grafana/). Por meio de uma interface de usuário é possível criar ou até mesmo importar dashboard disponibilizados pela comunidade. Na figura 20 temos um exemplo de dashboard construído no grafana.

O Grafana possui suporte para diversas aplicações como o PostgreSQL, Elasticsearch, MySQL e Prometheus.

Figura 20 –Grafana Dashboard



Fonte: https://grafana.com/api/dashboards/6417/logos/large

Referências

VALENTE, Marco Tulio. Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade, Editora: Independente, 395 páginas, 2020.

ARUNDEL, John.; DOMINGUS, Justin. DevOps nativo de nuvem com Kubernetes. São Paulo: Novatec, 379 páginas, 2019.