Container & Kubernetes

Trabalho final

Aluno: Marco Antonio Monteiro Pedro

Sumário

Relatório – Atividade Prática Containers & Kubernetes

1. Introdução

A atividade prática da disciplina **Containers & Kubernetes** tem como objetivo aplicar, em um cenário realista, os conceitos estudados sobre conteinerização com Docker, orquestração com Kubernetes, Deployments, Services e exposição de aplicações em cluster.
O desafio consiste em criar uma **prova de conceito (POC)** para uma empresa fictícia, demonstrando a execução de dois servidores web distintos (Nginx e Apache HTTPD) em um ambiente orquestrado com Kubernetes.

2. O que é YAML

YAML é uma linguagem de serialização de dados legível por humanos, usada principalmente para arquivos de configuração em sistemas modernos como Kubernetes, Docker e Ansible. Ele se destaca pela simplicidade, clareza e capacidade de representar dados estruturados de forma intuitiva.

2.1 Definição

- **YAML** significa originalmente *"YAML Ain't Markup Language"* (YAML não é linguagem de marcação).
- É um formato textual que descreve dados estruturados, semelhante ao JSON e XML, mas com foco em **legibilidade humana**.
- Usado para configurar aplicações, definir infraestrutura como código e automatizar processos.

2.2 Estrutura e Sintaxe

- **Indentação:** A hierarquia é definida por espaços (não usa chaves {} ou colchetes []).
- **Chaves e valores:** Representados como chave: valor.
- **Listas:** Usam o traço - para indicar itens.
- **Comentários:** Começam com #.
- **Extensões de arquivo:** .yaml ou .yml.

2.3 Exemplo simples:

```
app: websolutions
version: 1.0
services:
 - nginx
- apache
```

```
database:
 type: mariadb
 user: admin
 password: secret
```

2.4 Características principais

- **Legibilidade:** Fácil de escrever e entender, mesmo para quem não é programador.
- **Flexibilidade:** Suporta tipos de dados como strings, números, listas e mapas.
- **Compatibilidade:** Pode ser convertido facilmente para JSON.
- **Padronização:** Usado em diversas ferramentas de DevOps e infraestrutura.

2.5 Vantagens

- **Clareza:** Menos poluição visual que XML ou JSON.
- **Popularidade:** É o padrão de configuração em Kubernetes, Ansible e CI/CD pipelines.
- **Automação:** Permite descrever ambientes complexos de forma declarativa.
- **Integração:** Funciona bem com linguagens de programação e ferramentas de orquestração.

2.6 Casos de uso

- **Kubernetes:** Definição de Pods, Deployments e Services.
- **Docker Compose:** Configuração de múltiplos containers.
- **Ansible:** Playbooks para automação de infraestrutura.
- **CI/CD:** Configuração de pipelines em GitHub Actions, GitLab CI, etc.

2.7 Comparação rápida

Formato	Legibilidade	Uso comum	Sintaxe
YAML	Alta	Configuração infra (K8s, Ansible)	Indentação e listas
JSON	Média	APIs, web apps	Chaves {} e colchetes []
XML	Baixa	Documentos, sistemas legados	Tags

2.7 Conclusão

O YAML é essencial para quem trabalha com **DevOps, containers e orquestração**, pois permite descrever ambientes de forma clara e automatizável. Sua simplicidade e legibilidade tornam-no a escolha padrão em projetos modernos de infraestrutura.

2.8 Importância do YAML na orquestração de containers

- Permite descrever de forma declarativa os recursos do cluster.
- Garante reprodutibilidade e padronização das configurações.
- Facilita a manutenção e escalabilidade de aplicações.

2.9 Principais objetos Kubernetes

- **Pods:** Unidade mínima de execução, encapsula um ou mais containers.
- **Deployments:** Controla a criação e atualização de réplicas de Pods.
- **Services:** Exposição e balanceamento de carga entre Pods, garantindo acesso estável às aplicações.

2.10 Automação da solução proposta

1. Criação das imagens Docker personalizadas (Nginx e Apache).
2. Carregamento das imagens no Minikube.
3. Aplicação dos manifests YAML para criar Deployments e Services.
4. Verificação dos Pods e serviços ativos no namespace websolutions.
5. Acesso às aplicações via IP do Minikube nas portas configuradas.

3. Parte Técnica – Solução em YAML

3.1 Estrutura do projeto

```
websolutions/
├── README.md
├── docker/
│   ├── nginx/
│   │   ├── Dockerfile
│   │   └── html/
│   │       └── index.html
│   ├── apache/
│   │   ├── Dockerfile
│   │   └── html/
```

```
│ │   └── index.html
├── k8s/
│   ├── apache-deploy.yaml
│   ├── apache-svc.yaml
│   ├── mariadb-config.yaml
│   ├── mariadb-deploy.yaml
│   ├── mariadb-pvc.yaml
│   ├── mariadb-secret.yaml
│   ├── mariadb-svc.yaml
│   ├── namespace.yaml
│   ├── nginx-deploy.yaml
│   └── nginx-svc.yaml
```

3.2 Tecnologias utilizadas

- Docker
- Kubernetes (Minikube)
- Nginx
- Apache HTTP Server
- MariaDB
- Git e GitHub

3.3 Fluxo de execução

3.3.1. Build das imagens personalizadas:

```
docker build -t nginx-poc:latest -f docker/nginx/Dockerfile docker/nginx
docker build -t apache-poc: latest -f docker/apache/Dockerfile docker/apache
```

3.3.2. Carregar as imagens no Minikube:

```
minikube image load nginx-poc: latest
minikube image load apache-poc: latest
```

3.3.3 Aplicar os manifests Kubernetes:

minikube kubectl -- apply -f k8s/

3.3.4. Verificar os pods:

minikube kubectl -- get pods -n websolutions

3.3.5. Verificar o IP do Minikube:

minikube ip

3.3.6. Acessar os serviços:

Nginx: http://<IP_MINIKUBE>:8080

Apache: http://<IP_MINIKUBE>:8081

3.4 Acessando o MariaDB

Para testar a conexão com o banco de dados dentro do cluster:

minikube kubectl -- run -it --rm mariadb-client --image=mariadb:11.4 -n websolutions -- \

mysql -h mariadb-svc -uwebuser -pwebpass webdb

3.4.1 Observações sobre desligar a máquina

- Ao desligar o computador, o Minikube é parado e os pods deixam de rodar.

- As imagens Docker e os manifests permanecem salvos localmente.

- Os dados do MariaDB são mantidos graças ao PVC (PersistentVolumeClaim).

- Ao ligar novamente, basta executar:

minikube start

minikube kubectl -- apply -f k8s/

minikube kubectl -- get pods -n websolutions

3.5 Troubleshooting

- ErrImagePull → usar minikube image load para carregar imagens locais.

- ContainerCreating → verificar PVC ou ConfigMap.

- CrashLoopBackOff → checar probes e variáveis de ambiente.

3.6 Compatibilidade

Ferramenta	Versão testada	Observações
Docker	26.x (ex: 26.1.3)	Compatível com Minikube v1.37.0. Versões 28/29 causam erro de comunicação.
Minikube	v1.37.0	Funciona bem com Docker 25.x e 26.x. Para Docker 29.x é necessário Minikube >= v1.38.
Kubernetes	v1.30.0	Estável e suportado. Versão v1.34.0 apresentou falhas de inicialização.
Sistema	Debian 13 (Trixie)	Repositório oficial do Docker só fornece 28/29. Usar repositório Bookworm para instalar 25/26.

4. Entendendo o Fluxo da Aplicação

Para compreender como tudo funciona, vamos seguir o caminho desde a criação da imagem até o acesso pelo navegador:

4.1. *Dockerfile*

- É o roteiro que ensina o Docker a montar uma imagem.

- Nele definimos:

  - Qual imagem base usar (ex.: httpd:latest ou nginx:latest).

  - Quais arquivos copiar (HTML personalizado).

  - Qual porta o container deve expor.

- Resultado: uma *imagem Docker* pronta para rodar.

4.2. *Imagem Docker*

- É como uma "receita congelada": contém o servidor web + seus arquivos.
- Exemplo: apache-poc:1.0 e nginx-poc:1.0.
- A imagem é estática, não roda sozinha. Precisa ser instanciada.

4.3. *Deployment (Kubernetes)*

- Define como os pods devem ser criados e gerenciados.
- Específica:
  - Qual imagem usar.
  - Quantas réplicas (pods) rodar.
  - Labels para identificar os pods.
- Garante que os pods estejam sempre rodando e escaláveis.

4.4. *Pod*

- É a instância em execução da imagem dentro do cluster.
- Cada pod roda um container baseado na imagem (Apache ou Nginx).
- Se um pod cair, o Deployment recria automaticamente.

4.5. *Service*

- Cria um endereço fixo para acessar os pods.
- Usa *NodePort* para expor para fora do cluster.
- Exemplo:
  - Apache → http://<minikube-ip>:30081
  - Nginx → http://<minikube-ip>:30080

4.6. *Usuário no navegador*

- Digita a URL.
- O tráfego passa pelo Service → chega ao Pod → o Pod serve o HTML.
- Resultado: o usuário vê a página personalizada.

---

5. Fluxo resumido

[Dockerfile] → [Imagem Docker] → [Deployment] → [Pod] → [Service] → [Usuário]

---

6. Por que separar Apache e Nginx em Deployments e Services diferentes?

- *Modularidade*: cada aplicação tem sua própria configuração.

- *Escalabilidade independente*: é possível aumentar réplicas de um sem afetar o outro.

- *Manutenção facilitada*: se o Apache precisar de atualização, não interfere no Nginx.

- *Boas práticas do Kubernetes*: cada serviço deve ter seu próprio Deployment e Service.

7. Entendendo o Cluster Kubernetes

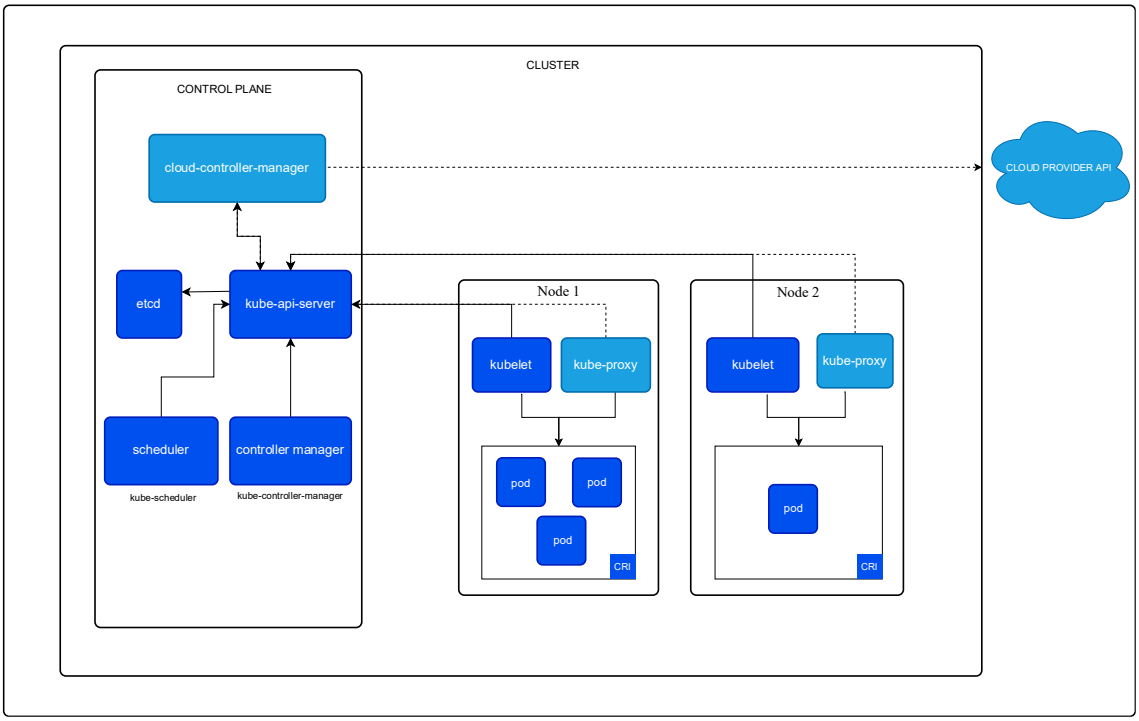
Um cluster Kubernetes é o conjunto de máquinas que trabalham juntas para rodar e orquestrar containers.

Ele garante que aplicações como Apache, Nginx e MariaDB rodem de forma organizada, escalável e acessível.

7.0.1 Arquitetura do Cluster

Um cluster Kubernetes consiste em um control plane mais um conjunto de máquinas trabalhadoras, chamadas de nodes, que executam aplicações conteinerizadas. Todo cluster precisa de pelo menos um worker node para executar Pods.

Os worker nodes hospedam os Pods que são os componentes da carga de trabalho da aplicação. O control plane gerencia os worker nodes e os Pods no cluster. Em ambientes de produção, o control plane geralmente executa em múltiplos computadores e um cluster geralmente executa múltiplos nodes, fornecendo tolerância a falhas e alta disponibilidade.



7.0.2 Componentes do control plane

Os componentes do control plane tomam decisões globais sobre o cluster (por exemplo, agendamento), bem como detectam e respondem a eventos do cluster (por exemplo, iniciar um novo pod quando o campo replicas de um Deployment não está satisfeito).

Os componentes do control plane podem ser executados em qualquer máquina do cluster. No entanto, para simplicidade, scripts de configuração normalmente iniciam todos os componentes do control plane na mesma máquina, e não executam contêineres de usuário nesta máquina. Consulte Criando clusters altamente disponíveis com kubeadm para um exemplo de configuração do control plane que executa em múltiplas máquinas.

7.0.3 kube-apiserver

O servidor da API é um componente da camada de gerenciamento do Kubernetes que expõe a API do Kubernetes. O servidor da API é o *front end* para a camada de gerenciamento do Kubernetes.

A principal implementação de um servidor de API do Kubernetes é o kube-apiserver. O kube-apiserver foi projetado para ser escalonado horizontalmente — ou seja, ele pode ser escalonado com a criação de mais instâncias. Você pode executar várias instâncias do kube-apiserver e distribuir o tráfego entre essas instâncias.

etcd

Armazenamento do tipo chave-valor consistente e de alta-disponibilidade, usado como armazenamento de apoio do Kubernetes para todos os dados do cluster.

Se o seu cluster Kubernetes usa o etcd como seu armazenamento de apoio, certifique-se de ter um plano de *backup* para seus dados.

Você pode encontrar informações detalhadas sobre o etcd na documentação oficial.

7.0.4 kube-scheduler

Componente da camada de gerenciamento que observa os Pods recém-criados e que ainda não foram atribuídos a um nó, e seleciona um nó para executá-los.

Os fatores levados em consideração para as decisões de alocação incluem: requisitos de recursos individuais e coletivos, restrições de hardware/software/política, especificações de afinidade e antiafinidade, localidade de dados, interferência entre cargas de trabalho, e prazos.

7.0.5 kube-controller-manager

Componente da camada de gerenciamento que executa os processos de controlador.

Logicamente, cada controlador está em um processo separado, mas para reduzir a complexidade, eles todos são compilados num único binário e executam em um processo único.

Existem muitos tipos diferentes de controllers. Alguns exemplos deles são:

- Node controller: Responsável por notar e responder quando nodes ficam indisponíveis.
- Job controller: Observa objetos Job que representam tarefas pontuais, depois cria Pods para executar essas tarefas até a conclusão.
- EndpointSlice controller: Preenche objetos EndpointSlice (para fornecer um link entre Services e Pods).
- ServiceAccount controller: Cria ServiceAccounts padrão para novos namespaces.

A lista acima não é exaustiva.

7.0.6 cloud-controller-manager

Um componente da camada de gerenciamento do Kubernetes que incorpora a lógica de controle específica da nuvem. O gerenciador de controle de nuvem permite que você vincule seu *cluster* na API do seu provedor de nuvem, e separar os componentes que interagem com essa plataforma de nuvem a partir de componentes que apenas interagem com seu cluster.

O cloud-controller-manager executa apenas controllers que são específicos do seu provedor de nuvem. Se você está executando o Kubernetes em suas próprias instalações, ou em um ambiente de aprendizado dentro do seu próprio PC, o cluster não tem um cloud controller manager.

Assim como o kube-controller-manager, o cloud-controller-manager combina vários loops de controle logicamente independentes em um único binário que você executa como um único processo. Você pode escalar horizontalmente (executar mais de uma cópia) para melhorar o desempenho ou para ajudar a tolerar falhas.

Os seguintes controllers podem ter dependências do provedor de nuvem:

- Node controller: Para verificar o provedor de nuvem para determinar se um node foi excluído na nuvem após parar de responder
- Route controller: Para configurar rotas na infraestrutura de nuvem subjacente
- Service controller: Para criar, atualizar e excluir load balancers do provedor de nuvem

---

7.0.7 Componentes do node

Os componentes do node executam em cada node, mantendo pods em execução e fornecendo o ambiente de runtime do Kubernetes.

7.0.8 kubelet

Um agente que é executado em cada nó no cluster. Ele garante que os contêineres estejam sendo executados em um Pod.

O kubelet utiliza um conjunto de PodSpecs que são fornecidos por vários mecanismos e garante que os contêineres descritos nesses PodSpecs estejam funcionando corretamente. O kubelet não gerencia contêineres que não foram criados pelo Kubernetes

O cluster é formado por duas partes principais:

7.1. PLANO DE CONTROLE (Control Plane)

- É o "cérebro" do cluster.

- Responsável por decidir o que deve rodar, onde e como.

- Componentes principais:

  - API Server → ponto de entrada para comandos (kubectl) .

- Scheduler → decide em qual nó cada pod vai rodar.

- Controller Manager → garante que o estado desejado seja mantido.

- etcd → banco de dados interno que guarda todo o estado do cluster.

7.2. NÓS (Nodes)

- São as "máquinas operárias" que realmente rodam os containers.

- Cada nó possui:

  - Kubelet → agente que conversa com o Control Plane.

  - Container Runtime → ex.: Docker ou containerd, que roda os containers.

  - Kube-proxy → cuida da rede e do roteamento de tráfego para os pods.

7.3. COMO FUNCIONA NO PROJETO WEBSOLUTIONS

- Usamos o Minikube, que cria um cluster local na máquina.

- Aplicamos os YAMLs → o Control Plane interpreta e entende o que precisa ser criado.

- O Scheduler decide em qual nó os pods vão rodar (no Minikube, há apenas um nó).

- O Kubelet cria os pods usando suas imagens Docker (apache-poc:1.0 e nginx-poc:1.0).

- O Service expõe esses pods para fora do cluster, permitindo acesso via navegador.

7.4. FLUXO RESUMIDO

[Usuário aplica YAML]

↓

[Control Plane interpreta]

↓

[Scheduler decide onde rodar]

↓

[Node cria Pods com containers]

↓

[Service expõe Pods para acesso externo]

↓

[Usuário acessa via navegador]

7.5. POR QUE PRECISAMOS DO CLUSTER?

- Escalabilidade → aumentar réplicas de Apache ou Nginx facilmente.

- Resiliência → se um pod cair, o cluster recria automaticamente.

- Isolamento → cada aplicação roda em seu próprio pod, sem conflito.

- Orquestração → o cluster garante que tudo esteja rodando conforme o desejado.

8. Cenário do Cliente

Imagine que o cliente chega até a Websolutions com a seguinte necessidade:

- Ele tem um site institucional (HTML simples) → hospedado no Nginx.

- Ele tem uma aplicação dinâmica (PHP ou outro backend) que precisa de banco → hospedada no Apache + MariaDB.

8.1. COMO O CLIENTE USARIA NA PRÁTICA

8.1.1. Entrega dos arquivos

- O cliente envia seus arquivos HTML/PHP para a Websolutions.

- Esses arquivos são colocados dentro das pastas:

  - docker/nginx/html

  - docker/apache/html

8.1.2. Empacotamento em imagem Docker

- A Websolutions gera uma imagem personalizada (nginx-poc:1.0 ou apache-poc:1.0) com os arquivos do cliente já embutidos.

- Isso garante que o site do cliente está pronto para rodar em qualquer ambiente.

8.1.3. Deploy no Kubernetes

- A imagem é carregada no cluster Minikube (ou em produção, num cluster Kubernetes real).

- O Deployment cria os pods e o Service expõe a aplicação.

8.1.4. Acesso ao site

- O cliente recebe uma URL/IP para acessar sua aplicação:

  - http://192.168.49.2:8080 → site no Nginx

  - http://192.168.49.2:8081 → aplicação no Apache

- Em produção, isso seria um domínio próprio (ex.: www.clienteA.com), apontando para o LoadBalancer do cluster.

8.1.5. Banco de dados (MariaDB)

- Se a aplicação precisar de persistência, o cliente usa o MariaDB já configurado no cluster.

- A Websolutions fornece credenciais seguras via Secret e configurações via ConfigMap.

8.2. EXPERIÊNCIA DO CLIENTE

Do ponto de vista do cliente, ele não precisa se preocupar com Docker ou Kubernetes. Ele apenas:

- Entrega os arquivos da aplicação.

- Recebe um endereço (URL/IP) para acessar.

- Se precisar, recebe credenciais para o banco.

Toda a parte de infraestrutura, escalabilidade e disponibilidade é responsabilidade da Websolutions.

9. Conclusão

A atividade prática permitiu consolidar os conceitos de conteinerização e orquestração, demonstrando como Kubernetes facilita a gestão de múltiplos serviços em um cluster. A prova de conceito evidencia a importância de ferramentas modernas para ambientes escaláveis e resilientes, alinhando teoria e prática em um cenário realista.

Fontes:

https://www.digitalocean.com/community/tutorials/a-rede-do-kubernetes-nos-bastidores-pt

https://kubernetes.io/pt-br/docs/concepts/architecture/