Insper

# Computação em Nuvem

Cap. 6 - CI/CD - 4 Aulas

Raul Ikeda - rauligs@insper.edu.br & Eduardo Marossi - eduardom12@insper.edu.br

**Grupo**:

Objetivos

1. Entender os conceitos básicos sobre Continuous Integration/Continuous Delivery (CI/CD).
2. Utilizar funções do Kubernetes como *deploy* de *pods* e gerenciamento de estado desejado.

Pré-requisitos:

1. Terminar o capítulo anterior (Container Orchestration)
2. Realizar a leitura sobre o Jenkins. [https://jenkins.io/].

Após a implantação do Kubernetes e uma tentativa de atualização de aplicação. Vamos montar um pipeline real onde a aplicação é montada e implantada automaticamente. Durante o processo ainda realizamos testes unitários e de integração. O ferramental utilizado nesse roteiro será: Kubernetes + Gitlab + Jenkins.

## Gitlab

- Conectar na instância *client* dentro do Openstack para executar comandos no k8s.
- Vamos utilizar a imagem Docker oficial para o Gitlab Community Edition: https://hub.docker.com/r/gitlab/gitlab-ce/
- Realizar um *deploy* da imagem do Gitlab Community Edition (gitlab/gitlab-ce:latest) na porta 80.

1. Transcreva o comando utilizado.

- Aguardar o serviço estar pronto:
  - $ watch -c kubectl get all
- Expor o serviço via LoadBalancer na porta 80. Aguardar. Não esquecer as portas.

2. Transcreva o comando utilizado.

## Jenkins

- Vamos utilizar a imagem Docker oficial LTS para o Jenkins: https://hub.docker.com/r/jenkins/jenkins/

- Ao invés de utilizar um comando como *kubectl run* para realizarmos um *deployment*, pode ser utilizado um arquivo de configuração *YAML* descrevendo toda a estrutura do *deployment*, *pods* e serviços que desejamos criar no k8s.
- Crie um arquivo jenkins.yaml contendo o template do *deployment*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: jenkins
labels:
  app: jenkins
spec:
replicas: 1
selector:
  matchLabels:
    app: jenkins
template:
  metadata:
    labels:
      app: jenkins
  spec:
    securityContext:
      runAsUser: 0
    containers:
    - name: jenkins
      image: jenkins/jenkins:latest
      ports:
      - containerPort: 8080
      - containerPort: 50000
      volumeMounts:
      - name: jenkins
        mountPath: /var/jenkins_home
      - name: docker
        mountPath: /var/run/docker.sock
      securityContext:
        privileged: true
    volumes:
    - name: jenkins
    - name: docker
      hostPath:
        path: /var/run/docker.sock
```

- Execute o comando:
- $kubectl apply -f jenkins.yaml
- Verifique se o *pod* do Jenkins inicia corretamente.
- Realize os mesmos passos realizados para o Gitlab, de forma a conseguir acessar o Jenkins pelo navegador.

3. Explique quais as vantagens e as desvantagens de utilizar um arquivo YAML para o *deployment* comparado ao utilizar o comando *kubectl create*.

## Configuração do Jenkins

- Verifique se o *pod* está rodando corretamente.
  - se não estiver, corrija.
- Acessar o *dashboard* principal do Jenkins.
- Seguir com a configuração fazendo a instalação recomendada dos Plugins.
- Coffe time!
- Criar um usuário *cloud* com a senha *cloud123*. Manter a URL padrão do Jenkins.
- Caso a página do Jenkins fique em branco após terminar a configuração ou realizar login. Altere o final do endereço no navegador para */restart* para forçar reiniciar o servidor do Jenkins.
- Caso esteja ocorrendo com frequência no Jenkins, o erro *No valid crumb was included in the request*, ir nas configurações do Jenkins, segurança global e habilitar a opção *Habilitar compatibilidade de proxy.*
- Entrar no painel de configuração do Jenkins e instalar os *plugins*:
- GitLab,
- GitLab Hooks
- Kubernetes Continuous Delivery
- Docker
- Entrar no painel de controle do Jenkins e criar um usuário *gitlab* e senha padrão com permissão de *Job/Build*
- Logar com este usuário e obter o *Token* da API para este usuário. Reserve.
- Vá novamente nas configurações do Jenkins e adicione uma nova nuvem do tipo *Kubernetes*. Configure a URL do Kubernetes seguindo o arquivo de configuração e Jenkins para o endereço correto do pod.

## Configuração do Gitlab

- Na página principal, configurar a nova senha do administrador para *cloud123*. Fazer o login.
- No painel de controle do Gitlab, vá em *General* e *Outbound requests* e habilitar para permitir acesso a rede local.
- No painel de controle, procure por *System Hooks*. Insira na URL, o endereço onde está localizado o seu Jenkins, assim como os seus dados de autenticação.
- Entre no painel de controle do Gitlab e crie um usuário com nome *jenkins* com acesso de Administrador.
- Crie um Token para o usuário, com nome jenkins com acesso *api*. Assim como associe uma senha para o usuário jenkins (cloud123)

## De volta ao Jenkins

- Novamente no painel, vá em Configurações Gerais e procure pelas configurações do Gitlab. No nome da Conexão utilize gitlab, insira o endereço do Gitlab. Adicione uma credencial do tipo Gitlab API para o Jenkins. Coloque o token gerado no Gitlab. Teste a conexão.
- Desmarcar "Enable authentication for '/project' end-point"
- Adicione também nas Configurações Gerais, uma nova Nuvem e escolha Kubernetes.
- Entre novamente na *interface Web* do Jenkins. Crie um novo *Pipeline* do Jenkins com nome *hello-node*, escolha a opção *Construir um projeto free-style*. Nas configurações detalhadas em seguida, escolha a conexão do Gitlab configurada anteriormente. Habilite a opção *Build when a change is pushed to GitLab*. Procure por *Pipeline Definition" e escolha "Pipeline script from SCM". Escolha SCM como Git e preencha a URL do seu repositório e especifique o script path para Jenkinsfile. Não esqueça no final de selecionar* Deploy to Kubernetes*

## De volta ao Gitlab

- Crie um repositório hello-node no Gitlab. Nas configurações do projeto, entre em Integrations e na URL coloque a URL do Jenkins com o projeto criado e o token do Jenkins. *http://192.168.5.230001/project/hello-node.*
- Crie um arquivo Jenkinsfile e inclua no seu repositório

```
podTemplate(label: 'pod-hello-node',
    containers: [
         containerTemplate(name: 'kubectl', image: 'smesch/kubectl',
              ttyEnabled: true, command: 'cat'),
    containerTemplate(name: 'docker', image: 'docker:stable-dind',
              ttyEnabled: true, privileged: true)],
        volumes: [secretVolume(secretName: 'kube-config', mountPath: '/root/.kube')],
        envVars: [containerEnvVar(key: 'DOCKER_TLS_CERTDIR', value: '')]
) {
    node ('pod-hello-node') {
        stage('Checkout') {
          checkout scm
        }

        container('docker') {
        stage('Docker Build & Push Current & Latest Versions') {
            sh ("docker login -u eduardom44 -p SUA_SENHA")
            sh ("docker build -t eduardom44/hello-world:${env.BUILD_NUMBER} .")
            sh ("docker push eduardom44/hello-world:${env.BUILD_NUMBER}")
            sh ("docker tag eduardom44/hello-world:${env.BUILD_NUMBER} eduardom44/hello-world:latest
            sh ("docker push eduardom44/hello-world:latest")
        }
        }

        container('kubectl') {
        stage('Deploy New Build To Kubernetes') {
            sh ("kubectl set image deployment/hello-world hello-world=eduardom44/hello-world:${env.B
        }
        }

    }
}
```

```
pipeline {
  environment {
    registry = "rikeda/aio-webserver"
    registryCredential = 'dockerhub'
    dockerImage = ''
  }
  agent {
    kubernetes {
      label 'promo-app'  // all your pods will be named with this prefix, followed by a unique id
      //idleMinutes 5  // how long the pod will live after no jobs have run on it
      yamlFile 'build-pod.yaml'  // path to the pod definition relative to the root of our project
      defaultContainer 'docker'  // define a default container if more than a few stages use it, wil
    }
  }
  stages {
    stage('Testing') {
      steps{
        container('python') {
          sh 'python senha_teste.py'
        }
      }
```

```
    }
    stage('Building image') {
      steps{
        script {
          dockerImage = docker.build registry + ":$BUILD_NUMBER"
        }
      }
    }
    stage('Pushing image') {
      steps{
        script {
          docker.withRegistry( '', registryCredential ) {
            dockerImage.push()
          }
        }
      }
    }
    stage('Removing local image') {
      steps{
        script {
          sh "docker rmi $registry:$BUILD_NUMBER"
        }
      }
    }
  }
}
```

- Testar mandando um push e vendo se o projeto da build automatico e deploy para o Kubernetes

## Aplicação

Faça uma aplicação em Flask que

## Teste Unitário

## Teste de Integração

## Mensagem final

Teste unitário Orquestrador de K8s Continuous Integration Continuous Deployment