Pipeline KK

Produzido por Daniel Delesderrier da Silva

Este documento está organizado da seguinte forma:

1. Objetivo:

Este documento descreve como desenvolver uma pipeline para o processo de CI (Continuous Integration) utilizando o projecto do GitHub.

Foi realizado um fork deste projecto : *https://github.com/benc-uk/nodejs-demoapp*

Pré-requisitos da Pipeline:

1. Ter um passo de análise de código usando SonarQube
2. Ter um passo de validação de testes unitários
3. Possuir testes em paralelo
4. Gerar imagem em Docker
5. Fazer push da imagem versionada para um repositório público
6. Não utilizar as GitHub Actions

Além disso, algumas características merecem destaque pois influenciam nas decisões e nas ferramentas:

- Código fonte em Node.js - foi utilizado o framework Express com templates EJS.
- Possui alguns testes prontos:
    - Testes Mocha em: src/tests/
    - Testes Postman em: src/package.json
- Possui Dockerfile na raiz do projecto.

2. Resumo da Estrutura da Pipeline

Baseado nas condições pré-estabelecidas, foi criado um *workflow* da pipeline:

1. Checkout do projecto no GitHub
2. Instalar as dependências do NodeJS
3. Executar testes em paralelo:
    a. Análise de código no SonarQube
    b. Testes Mocha
    c. Testes Postman
4. Gerar imagem em Docker usando Dockerfile
5. Fazer push da imagem versionada para o DockerHub

3. Principais Ferramentas

Teste de Código SonarQube

SonarQube é uma ferramenta que avalia a qualidade do código fonte. Ela utiliza um conjunto de ferramentas que analisam arquitetura, design, número de comentários, regras de linguagem, nível de complexidade, código duplicado, cobertura de testes no código, etc. Além de possuir um número bastante extenso de plugins.

Por que usar SonarQube?

É uma das soluções mais utilizadas para analisar a qualidade do código. Tem muita documentação na internet e é um projecto Open Source. Durante a pipeline, o desenvolvedor consegue visualizar um resumo dos números que indicam a qualidade do código. Se quiser, ele também pode acessar o dashboard do portal e olhar com mais detalhes esses números e alertas, facilitando a visualização e identificação dos problemas.

Teste de Código: Mocha

Mocha é um framework Javascript que roda em aplicações Node.js e no browser para realizar testes assíncronos de uma maneira simples e fácil.

Por que usar Mocha?

A maioria dos desenvolvedores Node.js usam o Mocha.js. É open-source e a sua documentação é rica e de fácil compreensão. Além disso, dentro de um ficheiro do projecto (em src/package.json) foi identificado scripts de testes usando o Mocha.

Teste de Código: Postman e Newman

Postman é uma ferramenta muito utilizada no mundo das APIs. Ela permite facilmente fazer requisições HTTP, executar scripts e testes. Possui uma versão de linha de comando

chamada Newman que permite a integração das coleções criadas no Postman com um sistema de integração contínua, como o Jenkins nos testes automáticos.

Por que usar Postman e Newman?

Foram encontrados testes criados pelo Postman dentro de um ficheiro do projecto (em src/tests/postman_collection.json). Por isso foi necessário instalar o Newman para executar estes testes dentro da pipeline do Jenkins.

Integração Contínua

É natural que existam colisões de código quando há vários desenvolvedores codificando módulos (micro serviços) de um mesmo projecto. A Integração Contínua atua exatamente antecipando tais situações, permitindo que a equipa reaja imediatamente e garanta uma evolução muito mais segura da aplicação.

Jenkins

É uma ferramenta de Integração Contínua e automatizada que traz diversos benefícios. Sua principal funcionalidade é construir o projecto por completo de forma automática, rodando os testes disponíveis a fim de detectar antecipadamente erros e reduzindo riscos.

Por que Jenkins?

É a solução mais utilizada de automatização de CI, tem muita documentação na internet e é um projecto Open Source.

Jenkinsfile e Pipeline Declarativa (Pipeline As Code)

Uma Pipeline Declarativa percorre uma sequência de blocos e define quais serão as etapas que o software irá passar. Desde o fetch do código no SCM, compilações, testes, até o deploy em ambientes.

O ficheiro Jenkinsfile, com o código da Pipeline Declarativa, é geralmente adicionado na raiz do repositório do projecto.

Por que utilizar o Jenkinsfile e Pipeline Declarativa?

Possibilita adotar o conceito de “Pipeline As Code”, permitindo armazenar todo histórico de alterações no Git e atualizar o código através de qualquer editor de texto. Ou seja, não precisará “abrir” o Jenkins para alterar a pipeline.

GitHub Webhooks e Build Triggers

Com o GitHub Webhooks é possível acionar automaticamente cada *build* no servidor Jenkins após cada *commit* no seu repositório Git. Uma outra opção é utilizar “Build Triggers > Consultar Periodicamente o SCM”. Isto criará um *scheduler* para realizar o *build* da aplicação.

Ele utiliza o padrão CRON do Linux (crontab). Por exemplo, “ * * * * * ” para ser executado a todo minuto.

Por que usar GitHub webhooks e Build Triggers?

Para automatizar o início da pipeline sem a necessidade do desenvolvedor acessar o Jenkins e manualmente “buildar” a Pipeline.

Plugins do Jenkins

Jenkins possui centenas de *plugins.* Alguns precisam de instalação para serem utilizadas no código Jenkinsfile e para a pipeline funcionar corretamente. Os principais *plugins* utilizados neste projeto são:

- Pipeline
- Docker Pipeline
- Sonarqube Jenkins
- Sonarqube Scanner
- Git

Por que utilizar plugins?

Os *plugins* simplificam os *scripts* da pipeline. Sem eles é preciso aprofundar o nível de detalhamento de algumas funções no Jenkinsfile.

Execução de forma paralela (Parallel Step)

O Jenkins oferece a possibilidade de ramificar seu pipeline em etapas paralelas.

Por que utilizar Execução Paralela?

A pipeline funcionando com os Steps em paralelo permite que o processo seja mais rápido. Por isso, os três testes (SonarQube, Mocha e Postman) foram preparados para funcionarem em paralelo.

Credentials no Jenkins

Algumas ferramentas solicitam informações (token, usuário, senha, etc) para conectar. Por isso, é recomendado criar Credentials no Jenkins. Abaixo alguns exemplos de sítios que precisam desta configuração:

- GitHub
- SonaQube Token
- Docker hub

Porque usar Credentials no Jenkins?

É indicado utilizar as Credentials do Jenkins para evitar passar informações sigilosas no código da pipeline.

4. Explicação das Linhas de Código da Pipeline (Jenkinsfile)

Todas as pipelines do tipo Declarative começam com um bloco chamado pipeline.

```
pipeline {
```

Stages e Steps

A seção Steps define um ou mais *steps* a serem executados em um Stage.

Na etapa abaixo, o *script* faz o *checkout* do código no GitHub utilizando o comando "checkout scm". O "scm" é uma variável especial que instrui a clonagem do repositório da Pipeline.

```
  stage('Clone repository'){

    steps {

      checkout scm

    }
  }
```

Script

Local definido para escrever trechos do tipo Scripted dentro da pipeline Declarative. O que é muito interessante, já que permite o DevOps trabalhar a lógica do script. Por exemplo, pode utilizar o "if" para fazer uma condição.

Abaixo vemos o script entrar no diretório "src" e, em seguida, executar o "npm install" que instala e atualiza todas as dependências do projecto em NodeJS.

```
  stage('Install dependencies') {
    steps {

      sh 'cd src && npm install'

    }
  }
```

Agora, damos início aos testes do código fonte. Como foi solicitado que os testes rodassem em paralelo, foi adicionado o comando "Parallel".

```
  stage('Run Tests in Parallel') {
    parallel {
```

O primeiro teste, como visto a seguir, é um teste de qualidade do código realizado pelo SonarQube. Foi utilizada a ferramenta sonar-scanner que é um scanner que faz a análise de código no SonarQube. Para isso, precisamos passar alguns parâmetros:

- -Dsonar.projectKey=delesderrier_nodejs-kk: -> Nome do projecto.
- -Dsonar.sources=src: -> Localização do código da aplicação.
- -Dsonar.host.url=http://10.0.0.100:9000: -> URL e porta do servidor do SonarQube.
- -Dsonar.login=8a4b975a864dbb3d6d3ea663a8d479b7f21b4cc6 -> Token de autenticaçao do Sonaqube. Ele é criado acessando a ferramenta SonarQube.

```
    stage('Tests Quality Check SonarQube') {
      steps {
       sh "/home/ddd/developer/sonar-scanner-4.6.0.2311-linux/bin/sonar-scanner \
           -Dsonar.projectKey=delesderrier_nodejs-kk \
           -Dsonar.sources=src \
           -Dsonar.host.url=http://10.0.0.100:9000 \
           -Dsonar.login=8a4b975a864dbb3d6d3ea663a8d479b7f21b4cc6"
      }
    }
```

Continuando, iniciam os testes npm configurados no ficheiro: src/package.json, na linha 20: "test": "mocha --exit ./tests"

```
    stage('Tests Mocha') {
      steps {
        sh 'cd src && npm test'
      }
    }
```

O Newman foi utilizado para rodar testes do Postman que estavam no ficheiro: src/tests/postman_collection.json.

O parâmetro "--suppress-exit-code 1" é necessário caso não queira que o resultado interrompa a pipeline. Ou seja, com este parâmetro adicionado, mesmo que algum teste do Postman falhe, a pipeline não será interrompida. Isso pode ser importante quando existem testes que não são considerados críticos.

```
    stage('Tests Postman-Newman') {
      steps {
                sh 'cd src && /usr/local/bin/newman run ./tests/postman_collection.json
--suppress-exit-code 1'
      }
    }
```

O *stage* abaixo cria a *buid* do Docker usando o Dockerfile que está na raiz do projecto do GitHub. Foi informado o repositório e o nome da imagem do docker ("delesderrier/nodejs-kk") .

```
  stage('Build image') {
    steps {
      script {

       docker = docker.build("delesderrier/nodejs-kk")
```

```
      }
    }
  }
```

No último *stage* é publicada a imagem no DockerHub, como pode ser visto a seguir.

O “withCredentials” para pegar o usuário e senha armazenados dentro do Credentials do Jenkins. Desta forma, não é necessário informar esses dados sigilosos no código da pipeline.

Em seguida, é executado o comando “docker login” para logar no DockerHub (usuário e senha mascarados).

Por fim, é executado o “docker push” para publicar a nova imagem versionada (delesderrier/nodejs-kk:latest) do docker no repositório público do DockerHub.

```
  stage('Push image') {
    steps {
      script {

                        withCredentials([usernamePassword(credentialsId:  'userpassdockerhub',
passwordVariable: 'dockerHubPassword', usernameVariable: 'dockerHubUser')]) {
          sh "docker login -u ${env.dockerHubUser} -p ${env.dockerHubPassword}"
          sh 'docker push delesderrier/nodejs-kk:latest'
        }

      }
    }
  }
```

5. Detalhes do Ambiente de Teste

Para não ficar só na teoria e testar o código do Jenkinsfile, foi criado um ambiente simplificado de teste utilizando um computador particular. Nele foi possível também reproduzir detalhes, vídeos e provas de conceito. O ambiente, basicamente, foi criado utilizando uma máquina virtual no VirtualBox e um container Docker.

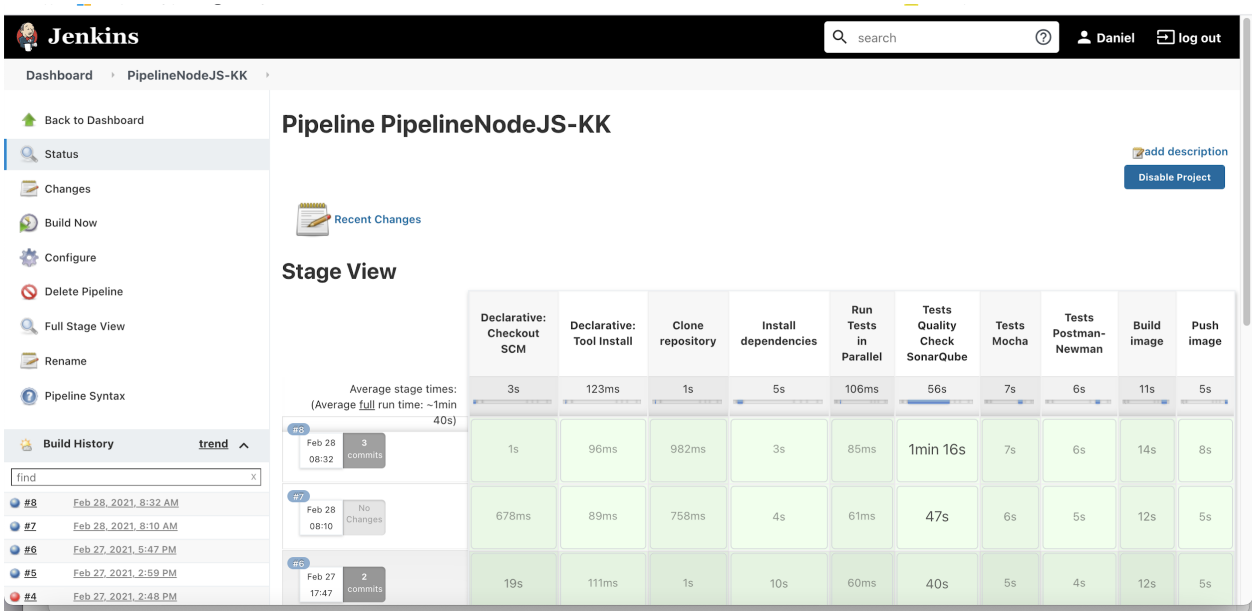
Jenkins Environment

O Jenkins foi instalado dentro de um VirtualBox com as seguintes configurações:

- Sistema Operacional: Ubuntu 20.04.1 LTS
- Memória: 8 GB de RAM
- Disco: 20 GB
- IP: 10.0.101

Também foi necessário instalar outras ferramentas como o Sonar Scanner, Docker, Git, npm, Java, unzip, etc.

Abaixo, é possível ver o Jenkins com a pipeline PipelineNodeJS-KK:



SonarQube Environment

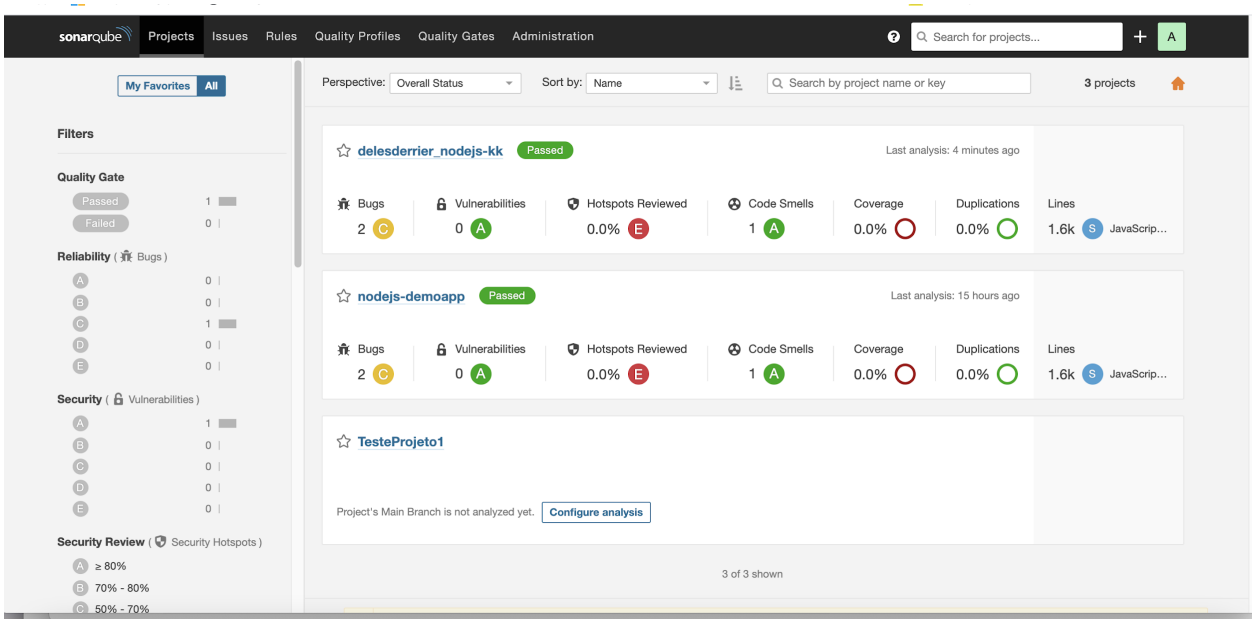
O SonarQube funcionou num container Docker com a imagem “sonarqube”, IP: 10.0.0.100, porta 9000 e utilizou-se as configurações docker-compose abaixo:

```
version: "3.2"

services:
  sonarqube:
    ports:
      - '9000:9000'
    volumes:
      - '/Users/danielsilva/developer/kk/data_dockeri/conf:/opt/sonarqube/conf'
      - '/Users/danielsilva/developer/kk/data_docke/data:/opt/sonarqube/data'
      - '/Users/danielsilva/developer/kk/data_docker/logs:/opt/sonarqube/logs'
      - '/Users/danielsilva/developer/kk/data_docker/extensions:/opt/sonarqube/extensions'
    image: sonarqube
```

Além de ter o portal de administração do SonaQube usado para criar o token, também podemos verificar o relatório detalhado da qualidade do código após a pipeline.

Print da análise de código do projecto “delesderrier_nodejs-kk” abaixo:



Daniel Delesderrier da Silva,
28 de fevereiro de 2021.