Jenkins

Fernando Anselmo
http://fernandoanselmo.orgfree.com/wordpress/

Versão 1.0 em 28 de junho de 2020

Resumo

Não é a mais intelectual das espécies que sobrevive; também não é a mais forte; mas a espécie que sobrevive é a única capaz de se adaptar melhor às mudanças no ambiente em que se encontra. (C. Megginson, interpretando Charles Darwin). **Jenkins** é uma poderosa ferramenta de código aberto destinado a executar Integração Contínua criada com a linguagem Java o que está associada a sua portabilidade para os mais diversos sistemas operacionais. Permite executar uma lista predefinida de etapas (denominado de *pipeline*), como por exemplo baixar o código-fonte de um repositório na Web, compilar conforme comandos da linguagem e construir um executável a partir das classes resultantes e publicá-la em um servidor definido. O gatilho para esta execução pode ser baseado em uma hora, um evento ou mesmo inciado por demanda.

1 Entrega Contínua

A medida que o número de trabalho a realizar aumenta, torna-se cada vez mais difícil para alguém mantê-los. Especialmente nos casos em que o trabalho é uma simples cópia que foi modificada a partir de um outro trabalho, acaba-se por tornar crucial manter uma determinada consistência. A entrega contínua (*Continuous Delivery* ou **CD**) é a prática de fornecer um software com mais qualidade e frequência. As práticas de CD podem incluir as seguintes vantagens:

- Promoção de código automatizado com qualidade.
- Estratégia de ramificação do projeto a ser entregue.
- Construções distribuídas e razoavelmente mantidas.
- Teste automatizado, distribuído ou paralelo.
- Provisionamento de um ambiente completamente atualizado e automatizado.

Sendo que um dos componentes fundamentais da CD é a Integração Contínua (*Continuous Integration* ou **CI**) em funcionamento. E é importante possuirmos um modelo de CI amplamente funcional. Principalmente se podemos considerar os seguintes parâmetros como: *pipeline* de CI codificáveis, automação na geração dos executáveis do projeto e ambientes de construção reproduzíveis com alta disponibilidade.

Um *pipeline* de CI é um conjunto de tarefas sequenciais ou paralelas (às vezes uma combinação de ambas). São configurados através de uma simples interface visual.

CD/CI é um processo no qual todo o trabalho de desenvolvimento se encontra integrado o mais cedo possível. Os artefatos resultantes são criados e testados automaticamente. Esse processo permite identificar erros em um estágio inicial do projeto. O **Jenkins**[1] é a ferramenta destina a fornecer toda essa funcionalidade.

Figura 1: *Logo do Jenkins*

Esta apostila não possui a pretensão de ensinar a usar o **Jenkins**[1], mas mostrar como usar um contêiner com o Jenkins e criar alguns *pipeline* de forma que possamos ter um ambiente totalmente funcional. Assim sedo podemos usá-la como um ponto de partida para compreendermos e colhermos os benefícios da CD/CI. Todos os comandos foram executados no sistema operacional Ubuntu.

2 Jenkins no Docker

Criar uma pasta que associará o contêiner:
```
$ mkdir $HOME/jenkins_home
```

Fornecer permissões a pasta de modo que o contêiner possa acessá-la:
```
$ chown 1000 $HOME/jenkins_home
```

Permitir o uso do arquivo Sock do Docker:
```
$ sudo chmod 777 /var/run/docker.sock
```

Baixar a imagem disponível:
```
$ docker pull jenkins/jenkins
```

Criar o container:
```
$ docker run --name meu-jenkins -d -v /var/run/docker.sock:/var/run/docker.sock
-v $(which docker):/usr/bin/docker -v $HOME/jenkins_home:/var/jenkins_home
-p 8081:8080 -p 50000:50000 jenkins/jenkins
```

Para executar abrir um navegador e acessar a URL `http://localhost:8081`.

2.1 Proceder a Instalação

Na primeira vez que acessamos o Jenkins devemos instalar o ambiente, verificar qual o Token do Jenkins para instalação:
```
$ docker logs meu-jenkins
```

[1] Jenkins é um software em constante evolução. Existem livros que auxiliam na administração do mesmo, aqui pretendo manter um padrão de simplicidade.

Localizar a linha:
`Jenkins initial setup is required.  An admin user has been created and a password generated.`
`Please use the following password to proceed to installation:`

`<<Número do TOKEN>>`

Após a criação do contêiner, ao acessar a URL `http://localhost:8081/`:

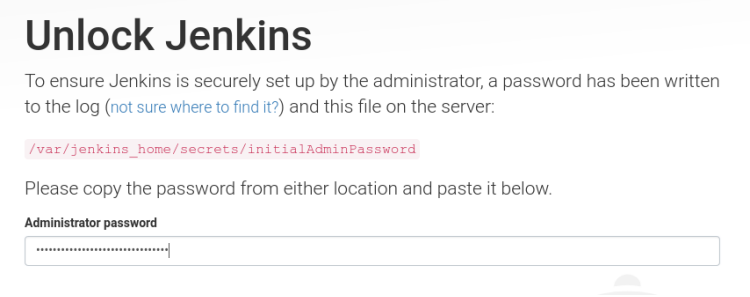

Figura 2: *Solicitação do Token*

Instalar os plugins sugeridos:

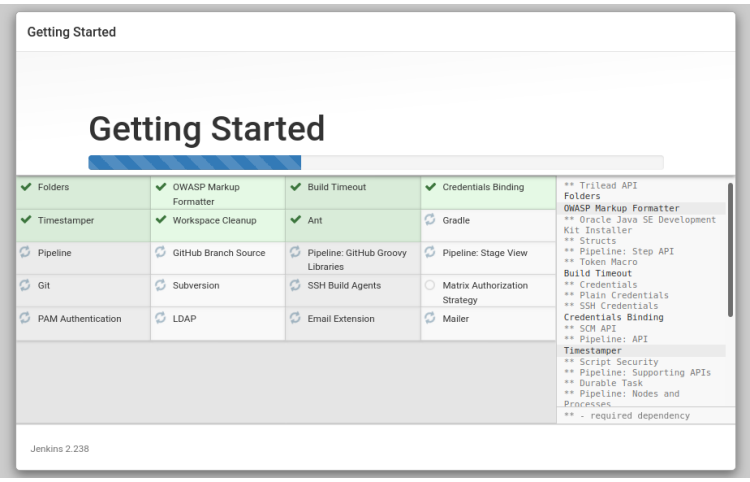

Figura 3: *Instalação dos plugins*

Escolher o usuário e senha, algo bem secreto como admin—admin:

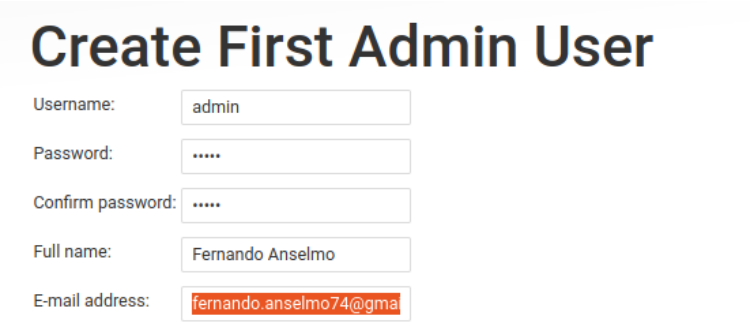

Figura 4: *Criação do Usuário e Senha*

Informar a URL do Jenkins e clicar no botão "Salvar e Finalizar":

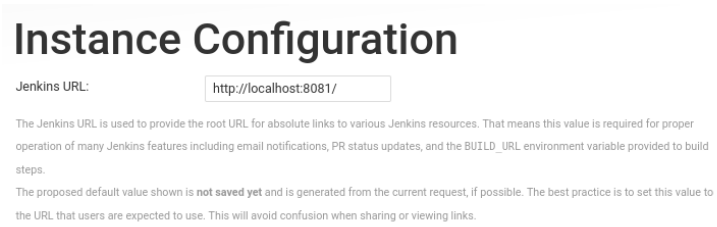

Figura 5: *Criação do Usuário e Senha*

Parabéns o Jenkins foi instalado com sucesso.

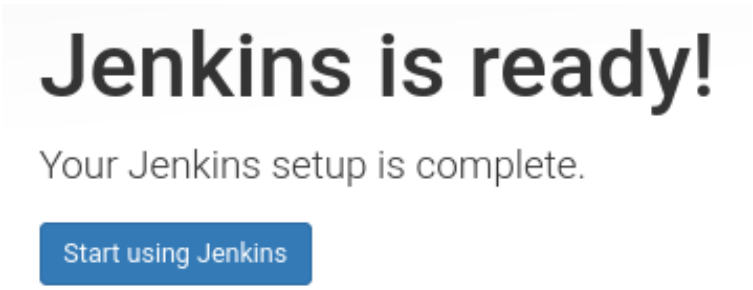

Figura 6: *Criação do Usuário e Senha*

3 Entrar no Jenkins

Algumas vezes precisamos de algumas informações relacionadas a máquina do Jenkins, então precisamos acessá-lo, no contêiner podemos fazer isso através do comando: // `$ docker exec -it meu-jenkins /bin/bash`

E estaremos dentro deste, podemos ver a versão do sistema: `$ cat /etc/os-release`

para sair digitamos o comando: `$ exit`

Este comando lista todos os contêineres que estão ativos: `$ docker container ls`

Observamos que cada contêiner possui um **CONTAINER ID**, este é usado, por exemplo para realizar uma inspeção de alguns detalhes deste: `$ docker container inspect [CONTAINER ID]`

3.1 Para atualizar o Jenkins no Docker

Atualizações de plugins no Jenkins podem ser realizadas direto no "Gerenciador do Jenkins", porém quando este informar que existe uma nova versão disponível, não é necessário baixá-la, apenas copiar o link do arquivo jenkins.war e proceder da seguinte forma:

Entrar no container do Jenkins:
`$ docker exec -u 0 -it meu-jenkins bash`

Baixar a última versão:
`$ wget [link do jenkins.war]`

Mover para o local correto:
`$ mv ./jenkins.war /usr/share/jenkins`

Mudar a permissão:
`$ chown jenkins:jenkins /usr/share/jenkins/jenkins.war`

Sair do bash:
`$ exit`

Reiniciar o container:
`$ docker restart jenkins`

4 Pipelines

Como já dissemos, um *pipeline* nada mais é do que uma sequencia de comandos que o Jenkins executará. A partir da versão 2.0 eles tomaram como base os arquivos script do Groovy. A documentação completa pode ser encontrada nesta URL `https://www.jenkins.io/doc/book/pipeline/syntax/`. Nesta apostila vamos ver alguns passos básicos para iniciarmos sem problemas. Por padrão possuem a seguinte estrutura:

```
pipeline {
  agent any
  stages {
    stage('Descritivo') {
      steps {
        // comandos
      }
    }
    ...
  }
}
```

Um script pode possuir vários estágios (*stage*) para executar e são sincronizados e dependentes, ou seja, não pode ocorrer erro no antecessor senão todo o processo será interrompido.

4.1 Hello World

Vamos criar um simples script para entendermos como esse conceito funciona. Na tela principal do Jenkins clicar em "New Item".

Na opção **Item Name** informamos **Hello World**. E escolhemos o modo **Pipeline**, e confirmamos ao clicar em **OK**.

Se já conhece o Jenkins notará que esta opção é bem simples e basicamente se concentra na seção **Pipeline** na qual escrevemos nosso script:

```
pipeline {
  agent any
```

```
  stages {
    stage('Hello') {
      steps {
      echo 'Hello World'
      }
    }
  }
}
```

Apply pode ser utilizado para verificar se existe qualquer problema na sintaxe. E uma vez terminado clicar em **Save**.

Para executar o script clicar em **Build Now**. Neste momento o Jenkins agenda seu script para ser executado assim que existir um agente livre. Pense que vários deles podem estar em execução neste momento, se assim ocorrer, e para não sobrecarregar a máquina, o Jenkins trabalha com sincronização e destina um serviço. Essa quantidade de serviços rodando simultaneamente pode ser configurada no Gerenciador.

A seguinte tela será mostrada caso seu script rode corretamente:

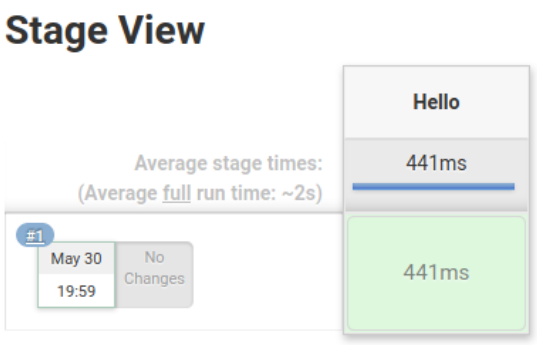

Figura 7: *Script Hello World executado com sucesso*

Ao clicarmos nessa área verde uma opção para visualizar o LOG será mostrada e a nossa mensagem aparecerá.

4.2 Hello World Docker

Sabemos que o Docker possui uma imagem de teste chamada **hello-world** e vamos usá-la para testar se está tudo OK. Proceda os mesmos passos descritos anteriormente para criarmos um novo *pipeline* chamado **Docker Hello** com o seguinte script:

```
pipeline {
  agent any
  stages {
    stage("Baixar Imagem") {
      steps {
        sh 'docker pull hello-world'
      }
    }
    stage("Executar Hello") {
      steps {
```

```
        sh 'docker run hello-world'
      }
    }
  }
}
```

E assim o Jenkins envia um comando ao Docker da máquina hospedeira (não dentro do contêiner) para trazer a imagem **hello-world** do repositório Docker e em seguida criar um contêiner. Observamos que esses estágios são totalmente dependentes pois se não conseguir trazer a imagem não teria sentido de criar um contêiner.

Após sua execução vamos dar o seguinte comando na máquina local:
`$ docker images`

Perceberemos agora a existência da imagem **hello-world** e o comando:
`$ docker ps -a`

Observamos que agora possuímos um novo contêiner já parado. Vamos agora executar o mesmo processo, porém ao término de mostrar a mensagem eliminar tanto o contêiner quanto a imagem.

Sabemos que para apagar um contêiner devemos dar o comando:
`$ docker rm [Contêiner ID]`

E para apagarmos a imagem: `$ docker rmi [Imagem ID]`

Apague ambos manualmente para não deixar nenhuma trilha no seu Docker. Lembre-se que o Jenkins só vai fazer o que pediu, então partimos da premissa que devemos conhecer todos os comandos que o Jenkins executará. Primeiro passo será conhecer o comando para localizar o ID do contêiner:
`$ docker ps -a --quiet --filter ancestor=[nome imagem]`

Que lista todos os IDs dos contêineres parados ou não (opção `-a --quiet`) que pertencem a uma determinada imagem (opção ancestor do filter).

A segunda parte do problema e próximo passo está em localizar o ID da imagem criada, que é resolvido com o comando: `$ docker images [nome imagem] --quiet`

Que lista apenas o ID (opção `--quiet`) de uma determinada imagem definida em [nome imagem].

Pronto, agora basta sabermos que no script podemos criar uma variável que armazena os IDs tanto do contêiner quanto da imagem para logo em seguida eliminá-los:
`variável = sh(script: "[comando]", returnStdout: true).trim()`

Sendo assim, modificamos nosso script para:

```
pipeline {
  agent any
  stages {
    stage("Baixar Imagem") {
      steps {
        sh 'docker pull hello-world'
      }
    }
    stage("Executar Hello") {
      steps {
```

```
        sh 'docker run hello-world'
      }
    }
    stage("Limpar") {
      steps {
        script {
          containerID = sh(script: "docker ps -a --quiet --filter ancestor=hello-world",
    returnStdout: true).trim()
          sh "docker rm ${containerID}"
          imagemID = sh(script: "docker images hello-world --quiet", returnStdout:
    true).trim()
          sh "docker rmi ${imagemID}"
        }
      }
    }
  }
}
```

Observamos que ganhamos mais um estágio. Nesse novo as ações devem estar dentro de uma tag *script* pois precisamos criar uma variável para conter nossos ID. Esse estágio "Limpar"poderia ser dividido em 2[2], ou seja, a criação de estágios busca uma melhor organização para seu *pipeline* e cabe ao Administrador resolver o resultado mais organizado[3].

4.3 Obter parâmetros

Parâmetros são essenciais para um pipeline, imaginemos que temos 4 ambientes: Desenvolvimento (DEV), Teste (TST), Homologação (HML) e Produção (PRD). Cada um desses ambientes define uma porta diferente para o Docker, uma solução seria criar 4 pipelines idênticos mudando somente o detalhe da porta (ou qualquer outro), porém devemos nos atentar na hora da manutenção e ter que corrigir erros em quatro pipelines (ao invés de um). Não seria então mais coerente criarmos um único pipeline e receber como parâmetro o ambiente?

```
pipeline {
  agent any
  parameters {
    string(name: "COMMIT", defaultValue: "*/development", description: "Informe o
    CommitID")
    choice(name: "AMBIENTE", choices: ["DEV", "TST", "HML", "PRD"], description:
    "Ambiente a executar")
  }
  stages {
    stage("Execução") {
      steps {
        script {
          if (params.AMBIENTE.equals("DEV")) {
            PORTA = "8080"
          }
          if (params.AMBIENTE.equals("TST")) {
            PORTA = "8081"
          }
          if (params.AMBIENTE.equals("HML")) {
            PORTA = "8082"
```

[2]Primeiro remove o Contêiner e em seguida a imagem.
[3]Compreenda que não existe uma receita de bolo.

```
19            }
20            if (params.AMBIENTE.equals("PRD")) {
21              PORTA = "8083"
22            }
23          }
24          echo "Aqui os dados: ${params.COMMIT} e ${params.AMBIENTE}"
25          echo "Pode usar assim: ${PORTA}"
26          echo "Ou assim: " + PORTA
27        }
28      }
29    }
30  }
```

Criamos para este pipeline dois parâmetros, o primeiro obtém o ID do Commit feito no GitHub (o último pode ser conseguido por "*/branch") e o segundo define em qual ambiente será executado. E se for, por exemplo, escolhido o ambiente de teste clicando na área verde temos:

Figura 8: *Saída do Pipeline*

Como este é um simples exemplo apenas mostramos no estágio o conteúdo dos parâmetros.

Um dado interessante: existe a opção "*This project is parameterized*"e assim que for salvo esta opção conterá os dois parâmetros aqui descritos. Então qual a vantagem de se colocar no script? É assim que está definido na documentação, então para não ocorrermos no perigo de em próximas versões essa opção desaparecer já sabemos como proceder.

4.4 Obter arquivos do Github

Nossos arquivos devem estar em um local aonde o Jenkins poderá buscá-los e proceder os passos necessários para transformá-los no artefato final. Devemos pensar na seguinte situação:

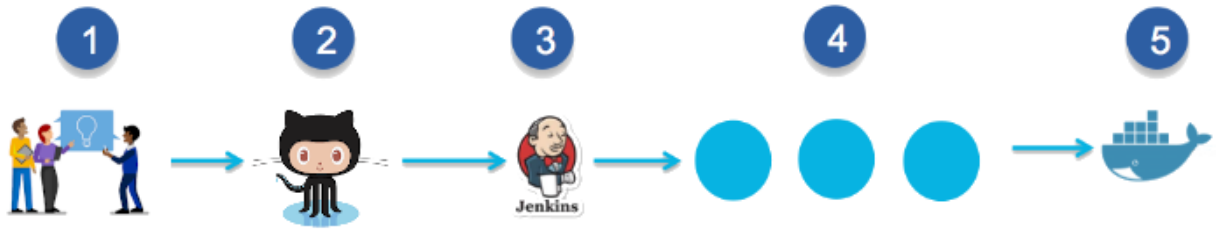

Figura 9: *Ciclo de Desenvolvimento*

No **passo 1** os desenvolvedores planejam e discutem como será o artefato a ser realizado, no **passo 2** sobem os códigos fontes necessários ao projeto para o GitHub, no **passo 3** o Jenkins obtém esses códigos e começar a processar o *pipeline* que está descrito no **passo 4** executando todos os processos necessários tais como compilação do código, execução do SonarQube para atestar a qualidade do

que foi disponibilizado está de acordo com as diretrizes da Empresa e ao término, no **passo 5** cria a imagem e publica o contêiner.

Podemos ver que não existe qualquer mágica nesse processo, somente a automatização de modo que se não tivéssemos o Jenkins para realizá-los seria manual ou através do uso de qualquer outra ferramenta que trabalhe de modo similar como o Bamboo, Buildbot, Apache Gump ou o Travis CI que são seus concorrentes mais diretos.

Considerando que já possui uma conta no GitHub de Administrador ativa, devemos informar para o Jenkins criar uma credencial, para isso, a partir da tela principal devemos acessar **Credentials** ▷ **System** ▷ **Global credentials** e selecionar a propriedade **Add Credentials**.

Em **Username** informar o usuário administrador do repositório, em **Password** sua senha e em **Description** a que se refere (não preencha o campo **ID**) e quando pressionarmos o botão OK um ID será criado para este registro. Salve-o pois o utilizaremos no script para acessar o GitHub.

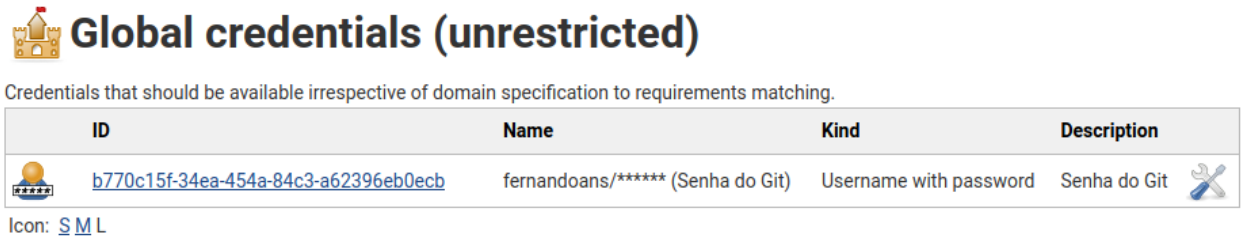

Figura 10: *Credencial criada*

Observamos que na imagem a credencial criada para meu usuário. Não tem medo de mostrá-la? Não, pois só é acessível pelo meu Jenkins que associa este número ao meu usuário e a senha[4]. Ou seja, mesmo que saiba esse número não lhe servirá absolutamente para nada. Por isso o Jenkins mostra que é uma área não restrita.

5 Criação de um projeto completo

Podemos criar qualquer tipo de projeto e para tentar deixar bem simples aqui faremos um Projeto JSP que mostrará uma página, como dissemos a mesma ideia pode ser aplicada a qualquer projeto, então usemos esse somente como um ponto de partida.

5.1 Construção do projeto

Usamos o Spring Tool Suite[2] para criar o projeto. File ▷ New ▷ Project..., na janela que se abre procurar por Web ▷ Dynamic Web Project. Clicar no botão Next. Informar o nome do projeto (por Exemplo **tstJenkins**), não esquecer de modificar a opção "Use an environment JRE"para a versão correta da Java Runtime desejada e pressionar o botão Finish. Ao término pedirá para mudar a perspectiva da janela para a visão J2EE. Se está tudo correto teremos a seguinte situação na aba *Project Explorer*:

[4]Como podemos perceber está bem escondida, então não se preocupe.

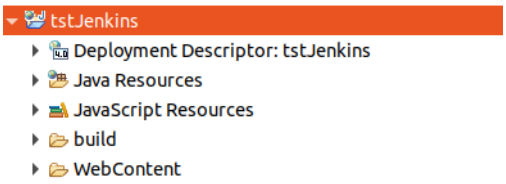

Figura 11: *Projeto Decus criado*

No projeto na pasta WebContent vamos criar um arquivo chamado "index.jsp"com o seguinte conteúdo:

```
<%@page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html lang="pt-BR">
<head>
 <title>Exemplo com Jenkins e Docker</title>
 <meta charset="UTF-8">
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
 <h1 style="color:black">JSP publicado com Jenkins</h1>
 Essa aplicação utiliza o Jenkins para realizar uma publicação em 4 fases:
 <ul>
  <li>Obter os códigos no GitHub</li>
  <li>Gerar o arquivo WAR com o Maven</li>
  <li>Criar um contêiner em enviá-lo para o DockerHub</li>
  <li>Executar o contêiner da aplicação</li>
 </ul>
</body>
</html>
```

Sim, poderia ser também "index.html"visto que não existe um único código JSP aqui, porém como dissemos antes, use este para um ponto de partida para qualquer projeto que pode ser desde a criação de *dashboard* para Análise de Dados até mesmo algo mais complexo. O objetivo desta apostila é mostrar os caminhos do Jenkins e não a concepção de projetos.

5.2 Construção da Imagem para a Aplicação

Sabemos que o Docker trabalha com imagens e o Jenkins necessita conhecer como é esse arquivo para poder gerá-la, na raiz do projeto devemos criar um arquivo chamado **Dockerfile**[5] com o seguinte conteúdo:

```
FROM tomcat:9
COPY target/*.war /usr/local/tomcat/webapps/tstJenkins.war
```

Como base usaremos a imagem do servidor **Apache TomCat 9**[6] e colocamos o arquivo **war** (*Web Archive*) no ponto correto para sua execução.

[5] Cuidado com as maiúsculas e minúsculas pois o nome deve ser exatamente este.
[6] Ou utilize outra de acordo com a configuração do servidor do seu projeto.

5.3 Apache Maven

O STS já é totalmente compatível com o Apache Maven[3] que é utilizado para a construção de todo o código então não precisamos nos preocupar muito com essa parte, porém o Jenkins precisa conhecer esse endereço, então devemos proceder sua instalação. Para isso basta baixar o arquivo compactado (apache-maven-3.6.3-bin.zip), descompactá-lo em uma pasta (a partir da raiz) e criar uma variável de ambiente chamada **mvnHome** de modo que podemos ter o acesso fácil a essa pasta.

Outro detalhe que precisamos é transformar o projeto para Maven de modo que o Jenkins possa realizar a compilação sem problemas. Clicar com o botão direito do mouse no projeto e acessar a opção: Configure ▷ Convert to Maven Project. Na janela apenas pressione o botão *Finish*. Se tudo está correto observamos que o projeto ganhou uma letra **M** o que indica agora é um projeto padrão Maven. Então foi criado um arquivo chamado **pom.xml**.

Neste arquivo adicionar as seguintes dependências:

```
<dependencies>
 <dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.0.1</version>
 </dependency>
 <dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
 </dependency>
</dependencies>
```

A primeira informa ao Maven que é um projeto JavaWeb e a segunda que deve ser adicionado o JUnit para a realização de testes unitários, é ideal que o Desenvolvedor se preocupe em garantir que todo o código entregue estará testado e funcionando para isso é essencial a implementação dos testes unitários.

5.4 GitHub

Obviamente precisamos de uma conta no GitHub para compartilhar o projeto (ou outro gerenciador de código). Clicar com o botão direito do mouse no projeto e acessar a opção: Team ▷ Share Project. Marcar a opção *Use or create repository in parent folder of project* e pressionar o botão **Create Repository**. Por fim o botão *Finish*.

Na perspectiva do Git é possível gerenciar todas as subidas e descidas para o repositório.

5.5 SonarQube

Com esta parte concluída podemos partir para a última fase, com o passar do tempo a complexidade do código tende a crescer, a realização de refatorações são sempre necessárias. Devemos nos preocupar em resolver os problemas de negócios, porém deixamos passar pequenos erros no código.

Para resolver problemas como esses devemos utilizar ferramentas que nos auxiliem na análise do código produzido.

O SonarQube é uma plataforma de código aberto para inspeção contínua da qualidade deste, para executar revisões automáticas com análise estática como forma de encontrar problemas, erros e vulnerabilidades de segurança que pode ser usado em mais de 20 linguagens de programação. Lembre-se que para usar essa funcionalidade no Jenkins o plug=in **SonarQube Scanner** deve estar instalado corretamente.

Baixar a imagem oficial:
`$ docker pull sonarqube`

Criar o contêiner:
`$ docker run -d --name meu-sonar -p 9000:9000 -p 9092:9092 sonarqube`

Acessar o SonarQube na URL `http:\\localhost:9000` com usuário e senha admin—admin. E deve ser aberta a seguinte janela:

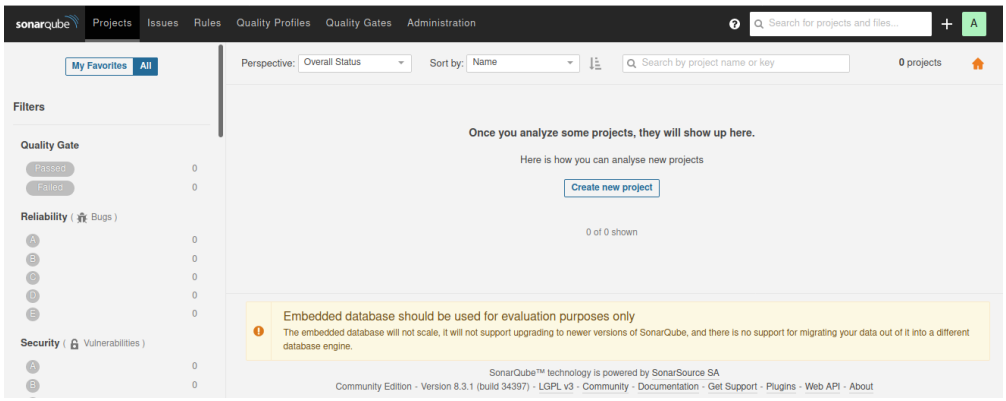

Figura 12: *Janela principal do SonarQube*

O botão *Create new project*. Ao clicar neste a solicitação para a chave do projeto, no qual podemos colocar o mesmo nome do projeto: tstJenkins. Assim fica mais fácil a sua localização. Em seguida a geração do Token, informar **meuToken**[7] e clicar no botão *Generate* e será gerado um código. Guardamos esse pois será necessário para o Jenkins.

Porém não usaremos tal funcionalidade para o Jenkins pois este exige outras configurações na qual recomendo consultar a documentação para verificar como proceder. Aqui manteremos as coisas simples, o STS possui o executor de comandos do Maven. Sendo que a meta completa que usaremos é a seguinte:

```
$ sonar:sonar -Dsonar.host.url=http://localhost:9000
-Dsonar.login=[numero do seu TOKEN] -Dsonar-projectName=tstJenkins
-Dsonar.projectVersion=master
```

Basta clicar com o botão direito no projeto e executar a Opção **Run As...**:

[7]Esse será salvo na seção do Administrador e pode ser usado por outros projetos - Guarde-o pois esse número NÃO SERÁ MAIS VISUALIZADO.

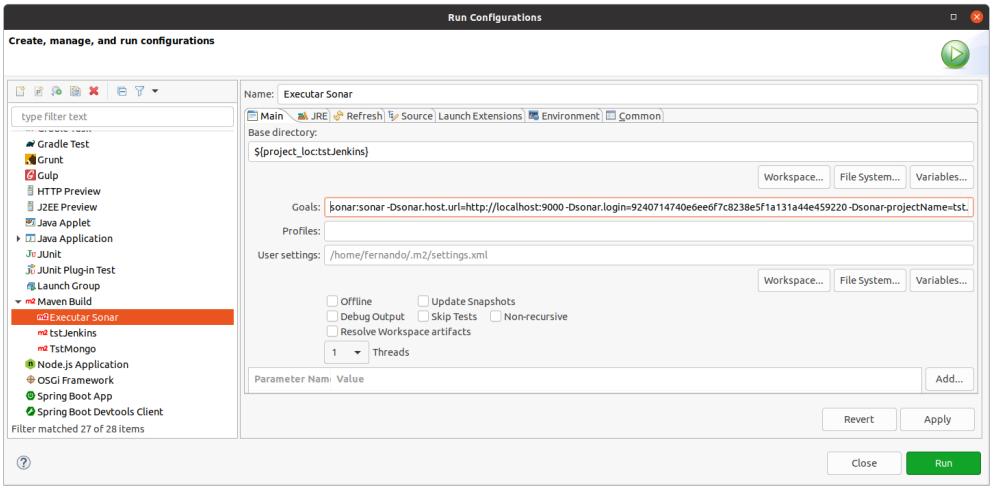

Figura 13: *Janela principal do SonarQube*

5.6 Pipeline Final

Com tudo posto podemos criar a seguinte pipeline:

```
pipeline {
 agent any
 stages {
  stage('Verificar Credenciais Git') {
   steps {
    git credentialsId: '[credencial]', url: 'https://github.com/fernandoans/tstJenkins'
   }
  }
  stage('construir') {
   steps {
    script {
     def mvnHome = tool name: 'Maven3', type: 'maven'
     def mvnCMD = "${mvnHome}/bin/mvn"
     sh "${mvnCMD} clean package"
    }
   }
  }
  stage('Construir Imagem Docker') {
   steps {
    script {
     sh 'docker build -t fernandoanselmo/meu-proj:1.0.0 .'
    }
   }
  }
  stage('Subir a Imagem') {
   steps {
    script {
     withCredentials([string(credentialsId: 'DockerHubPwd', variable: 'DockerHub')]) {
      sh "docker login -u fernandoanselmo -p ${DockerHub}"
     }
     sh 'docker push fernandoanselmo/meu-proj:1.0.0'
    }
   }
  }
```

```
35   stage('Executar Local') {
36    steps {
37    script {
38      containerID = sh(script: "docker ps --quiet --filter name=meu-proj", returnStdout:
     true).trim()
39      containerID = sh(script: "docker ps -a --quiet --filter name=meu-proj",
     returnStdout: true).trim()
40      if (!containerID.isEmpty()) {
41       sh 'docker stop meu-proj'
42      }
43      if (!containerID.isEmpty()) {
44       sh 'docker rm meu-proj'
45      }
46      sh 'docker run -p 8080:8080 -d --name meu-proj fernandoanselmo/meu-proj:1.0.0'
47     }
48    }
49   }
50  }
51 }
```

E como resultado final obtemos a seguinte imagem:

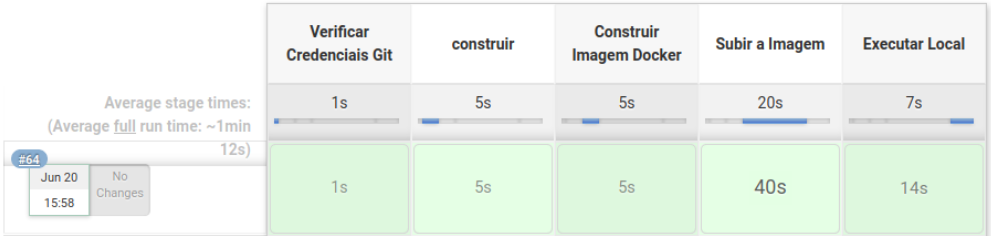

Figura 14: *Pipeline Final*

E ao acessar o endereço `http://localhost:8080/tstJenkins/` teremos uma tela como esta:

Figura 15: *Pipeline Final*

6 Conclusão

O conceito de CD/CI surgiu para remover os problemas de localizar ocorrências posteriores no ciclo de vida da construção e que os desenvolvedores que integram o código em um repositório

compartilhado em intervalos regulares possam ter a garantia que o mesmo estará em produção sem a interferência de falhas humanas na publicação.

A automação de compilações e testes, melhora drasticamente os ciclos de lançamento de novos artefatos. À medida que cresce, o setor criou um ciclo contínuo para as implantações de produção. Jenkins permite passar do check-in do código à implantação de uma nova versão do nosso aplicativo na produção sem causar qualquer estresse uma vez que todo processo foi checado e validado. Assim os desenvolvedores podem integrar facilmente as alterações de aplicativos com essa ferramenta para ajudar o usuário a obter uma nova versão.

Jenkins permite que o software seja testado e entregue continuamente com a ajuda de várias tecnologias de integração e implantação. Seus plugins nos ajudam a fornecer CD/CI em vários estágios diferentes e podem integrar qualquer ferramenta específica, como Git, Amazon EC2 ou Maven.

Sou um entusiasta do mundo **Open Source** e novas tecnologias. Qual a diferença entre Livre e Open Source? Livre significa que esta apostila é gratuita e pode ser compartilhada a vontade. Open Source além de livre todos os arquivos que permitem a geração desta (chamados de arquivos fontes) devem ser disponibilizados para que qualquer pessoa possa modificar ao seu prazer, gerar novas, complementar ou fazer o que quiser. Os fontes da apostila (que foi produzida com o LaTex) está disponibilizado no GitHub[7], assim baixar, alterar e usar. Veja ainda outros artigos que publico sobre tecnologia através do meu Blog Oficial[5].

Referências

[1] Site oficial do Jenkins
`https://www.jenkins.io/`

[2] Editor Spring Tool Suite para códigos Java
`https://spring.io/tools`

[3] Apache Maven
`https://maven.apache.org/`

[4] SonarQube
`https://www.sonarqube.org/`

[5] Fernando Anselmo - Blog Oficial de Tecnologia
`http://www.fernandoanselmo.blogspot.com.br/`

[6] Encontre essa e outras publicações em
`https://cetrex.academia.edu/FernandoAnselmo`

[7] Repositório para os fontes da apostila
`https://github.com/fernandoans/publicacoes`