



**GOVERNO DO
ESTADO DO
TOCANTINS**

CURSO DE SISTEMAS DE INFORMAÇÃO

APLICAÇÃO DE TDD NA HOSPEDAGEM DE SISTEMAS WEB UTILIZANDO DEVOPS

APOLYANE FARIAS LOPES

Palmas - TO

2018



**GOVERNO DO
ESTADO DO
TOCANTINS**

CURSO DE SISTEMAS DE INFORMAÇÃO

APLICAÇÃO DE TDD NA HOSPEDAGEM DE SISTEMAS WEB UTILIZANDO DEVOPS

APOLYANE FARIAS LOPES

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação da Universidade Estadual do Tocantins - UNITINS, como parte dos requisitos para a obtenção do grau de Bacharel em Sistemas de Informação, sob a orientação do professor Me. Douglas Chagas da Silva.

Palmas - TO

2018



**GOVERNO DO
ESTADO DO
TOCANTINS**

CURSO DE SISTEMAS DE INFORMAÇÃO

APLICAÇÃO DE TDD NA HOSPEDAGEM DE SISTEMAS WEB UTILIZANDO DEVOPS

APOLYANE FARIAS LOPES

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação da Universidade Estadual do Tocantins - UNITINS, como parte dos requisitos para a obtenção do grau de Bacharel em Sistemas de Informação, sob a orientação do professor Me. Douglas Chagas da Silva.

Prof. Me. Douglas Chagas da Silva
Orientador

Prof. Me. Alex Coelho
Convidado 1

Prof. Esp. Fredson Vieira Costa
Convidado 2

Palmas - TO
2018

Dados Internacionais da catalogação na publicação (CIP)
Biblioteca da Universidade Estadual do Tocantins
Campus Graciosa – Palmas - TO

L864a Lopes, Apolyane Farias
 Aplicação de TDD na hospedagem de sistemas WEB utilizando
DevOps / Apolyane Farias Lopes – Palmas - TO, 2018.
77 fls.; il.; col.

Inclui CD-ROM

Orientação: Prof^o. Douglas Chagas da Silva

TCC (Trabalho de Conclusão de Curso). Sistemas de Informação.
Universidade Estadual do Tocantins. 2018

1. Test Driven Development. 2. DevOps, 3. Testes de Software. 4.
Garantia de Qualidade. I. Silva, Douglas Chagas da II. Título. III.
Sistemas de Informação.

CDD: 004.06

Ficha catalográfica elaborada pela Bibliotecária – Maria Madalena Camargo – CRB
2/1527

Todos os Direitos Reservados – A reprodução parcial, de qualquer forma ou por
qualquer meio deste documento é autorizado desde que citada a fonte. A violação dos
direitos do autor (Lei nº 9.610/98) é crime estabelecido pelo artigo 184 do código penal.



UNITINS
Universidade Estadual do Tocantins



**GOVERNO DO
ESTADO DO
TOCANTINS**

108 Sul, Alameda 11, Lote 03, Caixa Postal 173, Palmas - Tocantins - Cep: 77020-122 - Tel: +55 63 3218-2939 - www.unitins.br

UNIVERSIDADE ESTADUAL DO TOCANTINS PRÓ-REITORIA DE GRADUAÇÃO

ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO

Aos seis dias do mês de dezembro, reuniu-se a banca examinadora do trabalho apresentado como Trabalho de Conclusão de Curso em **Sistemas de Informação** de **APOLYANE FARIAS LOPES**, intitulada: **APLICAÇÃO DE TDD NA HOSPEDAGEM DE SISTEMAS WEB UTILIZANDO DEVOPS**. Compuseram a banca examinadora os professores **Me. Douglas Chagas da Silva (Orientador)**, **Me. Alex Coelho** e **Me. Fredson Vieira Costa**. Após a exposição oral, o (a) candidato(a) foi arguido(a) pelos componentes da banca que reuniram-se reservadamente, e decidiram, APROVAR, com o conceito 9,5 a monografia. Para constar, redigi a presente Ata, que aprovada por todos os presentes, vai assinada por mim, Coordenadora de TCC do curso de Sistemas da Informação, e pelos demais membros da banca.



Me. Douglas Chagas da Silva



Me. Alex Coelho



Me. Fredson Vieira Costa



Coordenação de TCC/Sistemas da Informação

Ao meu estimado companheiro Luke, the cat.

Agradecimentos

Agradeço a Deus por seu cuidado com minha vida, por ter me concedido saúde e força para superar todas as dificuldades.

A Universidade Estadual do Tocantins, pela oportunidade de fazer o curso.

Ao professor Douglas Chagas, pela orientação com suas correções e incentivos. E também por seu apoio e confiança.

Agradeço a todos os professores que fizeram parte de minha formação, por me proporcionarem o conhecimento. Sempre ensinando com muita humildade e sabedoria. Serei eternamente grata.

Aos meus pais, irmãos e a toda a minha família que a distância, mas com muito carinho e apoio, sempre torceram para que eu chegasse até esta etapa de minha vida.

Aos meus amigos e colegas de universidade, que contribuíram para o meu desenvolvimento intelectual e pelas parcerias nos projetos e estudos.

Enfim, a todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

*"Deus não escolhe os capacitados, capacita os escolhidos.
Fazer ou não fazer algo só depende de nossa
vontade e perseverança."(Albert Einstein)*

Resumo

O avanço tecnológico oferece às organizações de Tecnologia da Informação uma boa oportunidade para renovação sistemática. Nesse contexto, o controle de qualidade e testes, que muitas vezes recebiam pouca atenção durante o desenvolvimento, passaram a ser realizados paralelamente ao ciclo de desenvolvimento. Com a adoção do *DevOps* e práticas ágeis, as equipes de controle de qualidade/testes podem beneficiar-se de um ambiente de *Test Driven Development (TDD)*, para garantir constantemente produtos de alta qualidade. Através dos *feedbacks* curtos e imediatos, é possível detectar antecipadamente as falhas, possibilitando sua correção imediata, caso contrário, a rapidez e a qualidade do *software* podem ser afetados. Este trabalho apresenta uma aplicação desta estratégia para implantação de sistemas *web* em um ambiente de hospedagem integrado.

Palavras-chaves: *Test Driven Development*, *DevOps*, Testes de *Software*, Garantia de Qualidade.

Abstract

The technological advance offers Information Technology organizations a good opportunity for systematic renewal. In this context, quality control and testing, which often received little attention during development, started to be carried out in parallel with the development cycle. With the adoption of DevOps and agile practices, quality control / testing teams can take advantage of a Test Driven Development (TDD) environment to consistently ensure high-quality products. Through short and immediate feedbacks, it is possible to detect faults in advance, allowing them to be corrected immediately, otherwise the speed and quality of the software can be affected. This work presents an application of this strategy for the deployment of web systems in a hosting environment.

Key-words: *Test Driven Development, DevOps, Software Testing, Quality Assurance.*

Lista de ilustrações

Figura 1 – Processo geral de <i>TDD</i>	10
Figura 2 – Abstração do conceito <i>DevOps</i>	12
Figura 3 – Modelo CAMS - Pilares do <i>DevOps</i>	13
Figura 4 – Fluxo pipeline <i>DevOps</i>	14
Figura 5 – Arquitetura de referência <i>DevOps</i>	14
Figura 6 – Arquitetura da Plataforma de Hospedagem	22
Figura 7 – Diagrama de Pacotes – UML	24
Figura 8 – Diagrama de Caso de Uso - UML	25
Figura 9 – Diagrama de Atividades - UML	25
Figura 10 – Diagrama de Classes - UML	26
Figura 11 – <i>Pipeline Jenkins</i>	30
Figura 12 – Gráfico de linha com o resultado dos testes unitários	34
Figura 13 – Gráfico de barra com o resultado dos testes funcionais	36
Figura 14 – Relatório geral da análise do SonarQube	37
Figura 15 – Confirmando acesso como administrador	52
Figura 16 – Tela de seleção de <i>plugins</i>	52
Figura 17 – Progresso da instalação de <i>plugins</i> sugeridos	53
Figura 18 – Tela criação de usuário administrador	54
Figura 19 – Tela de definição da URL do Jenkins	54
Figura 20 – Tela de finalização das configurações	55
Figura 21 – Tela inicial do Jenkins	55
Figura 22 – Página de Gerenciamento do Jenkins	56
Figura 23 – Página de Gerenciamento de <i>Plugins</i>	56
Figura 24 – Página de Criação de novo <i>Job</i>	57
Figura 25 – Página de Configuração do <i>Job</i>	57
Figura 26 – Página do <i>workspace</i> do <i>Job</i> recém criado	58
Figura 27 – Página de detalhes do <i>job</i>	58
Figura 28 – Página da saída do console	59
Figura 29 – Página Inicial do SonarQube	60
Figura 30 – Configurar SonarQube <i>Servers</i>	60
Figura 31 – Configurar SonarQube <i>Scanner</i>	61
Figura 32 – Configurar projeto com o SonarQube <i>Scanner</i>	62

Lista de tabelas

Tabela 1 – Classificação dos testes quanto ao papel nos processos (FILHO, 2009, p.352).	6
Tabela 2 – Recursos/Ferramentas Utilizados	20
Tabela 3 – Métricas de testes	23
Tabela 4 – Casos de teste e Testes unitários (PERCIVAL, 2017)	32
Tabela 5 – Casos de teste e Testes funcionais (PERCIVAL, 2017)	33
Tabela 6 – Definição das Métricas do SonarQube (S.A, 2018)	39
Tabela 7 – Relatório de <i>Bugs</i> SonarQube	39
Tabela 8 – Relatório de Vulnerabilidades SonarQube	40
Tabela 9 – Relatório de <i>Code Smells</i> SonarQube	40
Tabela 10 – Relatório de Duplicidades SonarQube	41
Tabela 11 – Relatório de Cobertura SonarQube	42

Lista de abreviaturas e siglas

BDD - *Behaviour Driven Development*

CAMS - *Culture, Automation, Measurement, Sharing*

CD - *Continuous Delivery*

CI - *Continuous Integration*

CVS - *Version Control System*

DBA - *Database administrator*

DEVOPS - *Development and Operations*

QA - *Quality Assurance*

TDD - *Test Driven Development*

TI - *Tecnologia da Informação*

Sumário

1	INTRODUÇÃO	1
1.1	Justificativa	2
1.2	Problema	3
1.3	Objetivos do Trabalho	4
1.3.1	Objetivo Geral	4
1.3.2	Objetivos Específicos	4
2	REFERENCIAL TEÓRICO	5
2.1	Testes de <i>Software</i>	5
2.1.1	Visão Geral	5
2.1.2	Teste de Unidade	7
2.1.3	Teste de Integração	7
2.1.4	Teste Automatizado	8
2.1.5	Test-Driven Development	9
2.1.5.1	Vantagens e Desvantagens do <i>Test-Driven Development</i>	9
2.2	<i>DevOps</i>	11
2.2.1	O que é <i>DevOps</i>	11
2.2.2	Pilares do <i>DevOps</i>	12
2.2.3	Terminologias <i>DevOps</i>	13
2.2.4	Qualidade de <i>Software</i> no ambiente <i>DevOps</i>	16
2.3	<i>TDD e DevOps</i>	17
2.3.1	Relação entre <i>TDD</i> e <i>DevOps</i>	17
2.3.2	Plataforma de Integração - Jenkins	17
3	METODOLOGIA	20
3.1	Tipo de Pesquisa	20
3.2	Materiais Utilizados	20
3.3	Descrição do Cenário de Testes	21
3.3.1	Métricas	23
3.3.2	Breve modelagem da aplicação modelo	24
3.3.3	Pseudocódigos dos estágios configurados	26
4	RESULTADOS	30
4.1	Resultados do <i>Pipeline</i> de Integração Contínua	30
4.1.1	Integração com git	31
4.1.2	Integração dos testes unitários	31

4.1.3	Integração dos testes funcionais	35
4.1.4	Integração com SonarQube	37
4.1.5	Integração com Docker para <i>build</i> da imagem da aplicação	43
5	CONSIDERAÇÕES FINAIS	44
5.1	Contribuições e Limitações	45
5.2	Trabalhos futuros	46
	REFERÊNCIAS	47
	APÊNDICES	50
	APÊNDICE A – INSTALAÇÃO E CONFIGURAÇÃO DO AMBIENTE . .	51
A.1	Instalando o Docker	51
A.2	Instalando o Jenkins via Docker	51
A.3	Configurando <i>Job</i> integrando Jenkins com Github	56
A.4	Instalando o SonarQube via Docker	59
A.4.1	Configuração SonarQube Scanner no pipeline	60
A.5	Script do <i>docker-compose</i> SonarQube	63
A.6	Scripts para containerização da aplicação	64

1 Introdução

Atualmente a cultura de Tecnologia da Informação (TI) está sofrendo grandes mudanças com relação a entrega de produtos e serviços. O foco está voltando-se para a entrega rápida de serviços, através da adoção de práticas ágeis e enxutas, no contexto de uma abordagem orientada para o sistema. Dentro dessa nova cultura, a filosofia do DevOps vem sendo adotada, uma vez que, de maneira geral pode ser entendida com a integração entre duas grades áreas: Desenvolvimento e Operações (KERSTEN, 2017).

O DevOps representa um movimento filosófico e cultural, onde é enfatizado um conjunto de boas práticas objetivando a colaboração e a comunicação dos profissionais de desenvolvimento (Dev) e operações (Ops), sob uma mesma cultura, práticas, ferramentas e linguagens. Visando acelerar as entregas do ciclo de vida de aplicativos e serviços, com elevado grau de qualidade (AIELLO; SACHS, 2014).

Desta forma, o movimento *DevOps* destaca bastante as práticas de automação de diversas atividades, de forma a entregar *software* de qualidade em produção, tais como: compilação do código, testes automatizados, criação de ambientes para teste ou produção, configuração da infraestrutura, monitoramento e métricas, desempenho, *deploy*, dentre outros (SATO, 2013).

Assim, melhorar a qualidade dos *softwares* entregues é um dos muitos benefícios que o *DevOps* pode oferecer, pois ao utilizar essa nova cultura tem-se: redução de falhas na implementação e aumento da frequência de entregas dentro do ciclo de desenvolvimento. Com o *DevOps* é possível ainda, a automatização desses processos, removendo o risco de falhas humanas que são comuns no modelo tradicional de desenvolvimento. Deste modo, é essencial criar uma cultura de colaboração entre pessoas com habilidades variadas (SATO, 2013).

Os testes asseguram que cada parte do sistema está funcionando como o esperado. De acordo com Myers (2004), o custo em correção cresce dez vezes para cada estágio que o projeto avança, isso significa que defeitos encontrados nas fases iniciais do desenvolvimento do *software* são mais baratos de serem corrigidos do que aqueles encontrados quando o *software* já está em produção.

Com o surgimento do manifesto ágil, onde seu objetivo é acelerar o desenvolvimento do *software* através de entregas de versões mínimas, os testes passaram a serem realizados em paralelo ao ciclo de desenvolvimento. Dessa forma, as funcionalidades são desenvolvidas e testadas por partes e continuamente (BECK, 2012).

Dentro das metodologias ágeis, uma abordagem que vem sendo muito utilizada é o *Test Driven Development (TDD)*, voltada para ajudar em todo o ciclo de desenvolvimento do *software*. Diferentemente das metodologias tradicionais, onde primeiramente o sistema é modelado, desenvolvido e por fim testado, no *TDD*, o teste é realizado do início ao fim do ciclo

de desenvolvimento, técnica conhecida como *test first* (PERCIVAL, 2017).

O relatório *State of DevOps*¹, publicado pela PUPPET (2017), comprova que as empresas que adotam as práticas de *DevOps* ultrapassam aquelas que não empregam. Ainda segundo esse relatório, organizações de alto desempenho em TI, automatizam significativamente mais seus processos de entregas de *software*, do gerenciamento de configuração aos testes, até implantações e aprovação de alterações. "O resultado é entregas de *softwares* mais rápido, mais confiável e com menos erros"(PUPPET, 2017, p. 26).

Diante do exposto, este trabalho objetiva realizar a aplicação de duas abordagens de desenvolvimento em um cenário de testes específicos, utilizando técnicas e conceitos *TDD*, no ciclo de integração contínua do *DevOps*, aplicado a um problema do mundo real.

O presente trabalho está organizado da seguinte forma: No capítulo 1 encontra-se a introdução e a justificativa, mostrando a problemática que será utilizada como base neste trabalho, e também os objetivos da pesquisa; no capítulo 2 apresenta-se um breve referencial teórico, apontando as principais tecnologias e conceitos envolvidos no processo de desenvolvimento do trabalho proposto; no capítulo 3 expõe-se a metodologia utilizada para resolução do problema; por fim, nos capítulos 4 e 5 são apresentados os resultados e considerações finais, respectivamente.

1.1 Justificativa

O avanço tecnológico oferece uma interessante oportunidade às organizações de TI para renovação processual. Assim como as empresas de bens materiais transformaram a forma como elas projetavam, criavam e disponibilizavam produtos usando a automação industrial durante o século 20, as empresas de hoje devem transformar como elas criam e disponibilizam *software* (AWS, 2018).

Para reduzir a incidência de problemas e aumentar a flexibilidade e a automação, é necessário a utilização de recursos automatizados. Neste sentido, as práticas de *DevOps* auxiliam as empresas no gerenciamento do lançamento de novas versões, disponibilizando atualizações mais rápidas e de maior confiabilidade aos seus clientes e usuários.

O processo de executar testes automatizados permite obter um *feedback* rápido sobre os riscos associados a uma versão de *software*. O uso de testes contínuos ajuda organizações a obterem uma vantagem competitiva acelerando entrega e minimizando riscos e custos de testes de *software*, obtendo impactos positivos no mercado (PUPPET, 2017).

Em (PRESSMAN, 2011) é apresentado que o custo com defeito na fase de engenharia de requisitos custa "um" enquanto encontrar o defeito durante a fase de produção custa cem vezes mais, então a utilização de teste, reduz os custo e não o contrário.

¹ Disponível em: <<https://puppet.com/resources/whitepaper/state-of-devops-report>>

A falta de teste de *softwares* não somente sai caro, como também causa prejuízos expressivos com sua correção, segundo afirma Aniche e Cardoso (2015) em:

"Os Estados Unidos estimam que *bugs* de *software* lhes custam aproximadamente 60 bilhões de dólares por ano. O dinheiro que poderia estar sendo usado para erradicar a fome do planeta está sendo gasto em correções de *software* que não funcionam."(ANICHE; CARDOSO, 2015, p.16).

Desse modo, a integração contínua, tem como objetivos identificar e investigar problemas rapidamente, melhorar a qualidade do *software* e reduzir o tempo que leva para validar e lançar novas atualizações de *software*. Isso pode garantir a simplificação dos processos no ciclo de melhoria da qualidade de *software* (SATO, 2013).

1.2 Problema

O objeto de avaliação deste trabalho constitui-se do seguinte questionamento: Como a prática de *TDD* pode trazer benefícios no contexto *DevOps* e como essas metodologias se relacionam?

Uma equipe de *software* focada no *DevOps* passa a beneficiar-se quando considera o controle de qualidade como parte integrante do desenvolvimento de aplicativos com alta qualidade; Tendo em vista que no cenário tradicional de desenvolvimento os testes de *software* e o controle de qualidade (*QA*) muitas vezes, recebem pouca atenção durante o desenvolvimento (MEZAK, 2017).

As equipes de *QA*/testes tem na cultura *DevOps* suporte para automatizar esse processo, desenvolvendo testes de qualidade em cada etapa do processo de desenvolvimento. Em um ambiente de desenvolvimento orientado a testes é possível observar se o código realmente funciona a cada passo dado. O rumo da programação se torna previsível através de ciclos de *feedback* curtos e imediatos.

Desta forma, a integridade e qualidade do código é constantemente garantida (MEZAK, 2017). Os especialistas da *DevOps* Aiello e Sachs (2014), em um relatório da IBM, afirmam que o *software* de qualidade requer não apenas uma mudança de comportamento, mas também uma mudança dramática na cultura organizacional.

Neste sentido, a prática de *TDD* vem auxiliar os desenvolvedores a sair do comodismo de escrever o código de produção - executar - debugar - consertar, para adotar uma nova maneira de testar códigos com eficiência.

1.3 Objetivos do Trabalho

1.3.1 Objetivo Geral

Avaliar a aplicação das metodologias do Desenvolvimento Guiado por Testes (*TDD*) no contexto de implantação e hospedagem de sistemas *web* utilizando a filosofia *DevOps*.

1.3.2 Objetivos Específicos

- Compreender os fatores que dificultam o alcance das metas estabelecidas para a obtenção da qualidade de testes envolvidos;
- Analisar os aspectos que facilitem ou prejudiquem a obtenção dos níveis da garantia de qualidade dos testes;
- Demonstrar fatores que reduzam as taxas de defeitos com a implantação de testes contínuos;
- Compreender a aplicação da metodologia de teste *TDD* no contexto de *DevOps*;

2 Referencial Teórico

A metodologia de testes consistirá em um cenário que vise a implantação de um sistema *web* de maneira funcional e transparente para o usuário, desta forma considerando aspectos como dependências, tratamento de arquivos estáticos e dinâmicos, permissões e *path's* da aplicação, versionamento e configuração da base de dados, dentre outros aspectos inerentes a automação da implantação dessas aplicações.

Desta maneira, ao longo deste capítulo serão apresentados os conceitos necessários ao entendimento das tecnologias, estratégias e ferramentas que serão utilizadas neste trabalho.

2.1 Testes de *Software*

Esta seção apresenta conceitos sobre Teste de *Software*, assim como os conceitos acerca do Teste Automatizado, Teste de Unidade e de Integração. A técnica de teste *TDD* é apresentada, evidenciando suas vantagens e desvantagens.

2.1.1 Visão Geral

Testes constitui o processo de avaliar se algum *software* está fazendo aquilo a que foi idealizado. Ainda de acordo com Pressman (2011), teste consiste em um conjunto de atividades a serem planejadas e executadas com antecedência durante o projeto. O principal objetivo dos testes de *software* é a localização de falhas, erros, defeitos e a verificação das funcionalidades do *software* (PRESSMAN, 2011).

O processo de testes de *software*, segundo Filho (2009), "visa verificar os resultados da implementação, através do planejamento, desenho e realização das atividades desse processo". Os testes, mais do que meios de detecção e correção de erros, são indicadores da qualidade do produto. Quanto maior o número de defeitos encontrados em um *software*, possivelmente maior ainda é a chance de ter defeitos não detectados (FILHO, 2009).

Ainda segundo Filho (2009), teste possui alguns elementos que devem ser observados durante o planejamento:

"Um teste pode ser visto como uma coleção de procedimentos e casos teste. Um procedimento de teste é um conjunto detalhado de instruções para execução de testes. Um caso de teste é uma especificação das entradas, resultados previstos e condições de execução para um item a testar."(FILHO, 2009, p.350).

O uso de teste tem como objetivo maximizar a cobertura do código, com objetivo de detectar a maior quantidade possível de defeitos que não foram encontrados durante a revisão do projeto (FILHO, 2009).

A Tabela 1, mostra uma lista dos principais tipos de testes classificados quanto ao papel que desempenham nas diversas etapas do ciclo de desenvolvimento de *software*.

Tabela 1 – Classificação dos testes quanto ao papel nos processos (FILHO, 2009, p.352).

Termo	Acrônimo em inglês	Definição
Teste de aceitação	<i>Acceptance testing</i>	Teste formal realizado para determinar se um sistema satisfaz a seus critérios de aceitação e capacitar um usuário, cliente ou outra entidade autorizada a determinar se aceita ou não o sistema.
Teste de desenvolvimento	<i>Development testing</i>	Teste formal ou informal realizado durante o desenvolvimento de um sistema ou componente, geralmente pelo desenvolvedor.
Teste de integração	<i>Integration testing</i>	Teste no qual componentes são combinados e avaliados para testar a interação entre eles.
Teste de qualificação	<i>Qualification testing</i>	Teste realizado para determinar se um sistema ou componente é adequado para uso operacional.
Teste de regressão	<i>Regression testing</i>	Teste seletivamente repetido de um sistema ou componente para verificar se alterações não causaram efeitos indesejáveis e se o sistema ou componente mantém a conformidade com seus requisitos especificados.
Teste de sistema	<i>System testing</i>	Teste conduzido em um sistema completo integrado, para avaliar sua conformidade com os requisitos especificados.
Teste de unidade	<i>Unit testing</i>	Teste individual de unidades ou grupos de unidades.
Teste funcional	<i>Functional testing</i>	Teste realizado para avaliar a conformidade de um sistema ou componente com os requisitos funcionais especificados.
Teste operacional	<i>Operational testing</i>	Teste realizado para avaliar um sistema ou componente em seu ambiente operacional.

Para realização deste projeto a principal distinção será feita entre os testes de unidade e de integração, classificados como testes de desenvolvimento, visto que os procedimentos de verificação são realizados pelos próprios desenvolvedores durante a implementação do código,

assim é possível determinar se os resultados das tarefas satisfazem aos objetivos que pretendiam atingir (FILHO, 2009).

2.1.2 Teste de Unidade

Ao desenvolver um *software*, cada componente do sistema é testado isoladamente. Esse teste conhecido como teste de unidade, verifica se o componente funciona adequadamente de acordo com os tipos de dados de entrada (PFLEEGER, 2004).

Segundo Pressman (2011), os testes de unidade foca na verificação da menor unidade testável de um projeto de *software* (componente ou módulo de *software*). Módulos são testados para descobrir erros dentro da limite estabelecido para o teste de unidade.

Esse teste pode ser realizado em paralelo com vários componentes e sua estrutura de dados é examinada para garantir que os dados avancem corretamente interna e externamente para a unidade de programa que está sendo testada (PRESSMAN, 2011).

Os testes de unidade pode auxiliar no processo de codificação, podendo ocorrer antes mesmo de começar a programar ou depois que o código estiver pronto. Cada caso de testes deverá ser acompanhado de um conjunto de resultados esperados (PRESSMAN, 2011).

Geralmente esse tipo de teste é utilizado no modelo tradicional ao final da codificação, o que pode ocasionar alguns problemas, principalmente no que se refere a rápida solução dos defeitos encontrados. Porém se os testes unitários forem bem planejados e executados podem encontrar cerca de 70% dos defeitos que poderiam ser encontrado por usuários (FILHO, 2009).

Quando os componentes do sistema já tiver sido testado por unidade, a próxima fase é assegurar que as interfaces entre as unidades trabalhem adequadamente integrados entre si. Para isso o teste de integração é aplicado para tal verificação (PFLEEGER, 2004).

2.1.3 Teste de Integração

Nesta técnica de teste, as unidades codificadas e verificadas por testes unitários são submetidas a um conjunto de testes de integração (FILHO, 2009).

O teste de integração, segundo Pressman (2011), é uma técnica utilizada com o objetivo de construir a estrutura completa do programa, integrando os componentes que foram testados em unidades e ao mesmo tempo testando, para descobrir possíveis erros associados com a interface.

Pressman (2011), nos apresenta duas formas para realizar a integração, a primeira é a não incremental, onde se constrói o programa inteiro, combinando todos os componentes previamente e é testado como um todo. Mas não parece ser é uma boa prática, pois é encontrado vários erros de uma vez, dificultando assim a correção pela falta de isolamento das causas (PRESSMAN, 2011).

Existe também a integração incremental, que é o oposto da estratégia anterior. O programa é construído e testado em pequenos incrementos, onde os erros são mais fáceis de serem identificados, isolados e corrigidos. Dentro dessa integração, possui duas abordagens que podem ser aplicadas, são elas (PRESSMAN, 2011):

1. Estratégia de integração descendente - Os módulos são integrados hierarquicamente, começando pelo módulo de controle principal (programa principal) e deslocando para baixo os módulos subordinados a ele. Dessa maneira os testes são feitos a medida que novos componentes são integrados. Essa estratégia verifica os principais pontos de controle ou decisão que fica nos níveis superiores da hierarquia.
2. Estratégia de integração ascendente - A construção começa nos níveis mais baixos da estrutura do programa, sua implementação segue os seguintes passos: (1) Componentes de baixo nível são combinados em *clusters*, que executam uma subfunção específica de *software*; (2) Um programa de controle de teste é escrito para coordenar entrada e saída do caso de teste; (3) O *cluster* é testado; (4) Os *drivers* são removidos e os *clusters* são movidos para cima na estrutura do programa.

Por outro lado, Filho (2009) apresenta a abordagem *big-bang*, semelhante a integração não incremental mencionada por Pressman (2011), onde as unidades são integradas de uma só vez. E as abordagens *top-down* e *bottom-up*.

Na integração *top-down* os testes são realizados na unidade principal no nível superior da arquitetura e a cada nova unidade adicionada é acrescentada funcionalidade real ao sistema. Já na integração *bottom-up* os testes são feitos nos níveis inferiores da arquitetura, onde são usado controles para simular funções das unidades de nível superior. Ao decorrer da integração esses controladores são substituídos por funcionalidades (FILHO, 2009).

2.1.4 Teste Automatizado

Testes automatizados são aqueles que são acionados por uma ferramenta que gera as entradas de teste por meio de controladores de teste. As ferramentas de automação podem usar programas ou *scripts* simples, os quais fazem verificações de forma automática nas funcionalidades do sistema que está sendo testado (FILHO, 2009).

Um dos principais benefícios desta abordagem é que os casos de testes podem ser refeitos a qualquer momento, seja em uma nova funcionalidade ou alguma modificação em uma funcionalidade específica. Sua implementação reduz os esforços e o tempo gasto, garantindo que não ocorrerá falha humana na execução dos testes (BERNADO; KON, 2008).

Nos processos atuais de desenvolvimento de *software* a automação de testes tornou-se indispensável, pois apenas esse tipo de teste pode garantir uma boa cobertura, executando de forma confiável grande volume de casos de testes (FILHO, 2009).

O surgimento das metodologias ágeis fez com que a atenção das equipes volte-se cada vez mais para a automação de tarefas manuais, dentre elas, a realização de testes de *software*. Os testes automatizados surgiram para melhorar a qualidade da análise e o tempo de execução dos testes, tornando o processo mais rápido e possibilitando maior cobertura do *software* (SOMMERVILLE, 2011).

Para Pressman (2011), o uso de algumas ferramentas tende a diminuir o tempo despendido para o teste, por isso são muito valiosas.

2.1.5 Test-Driven Development

Test-Driven Development faz parte dos princípios ágeis de desenvolvimento. Tal técnica foi criada por Kent Beck que impulsionou a ideia de escrever testes automatizados antes da implementação do código (BECK, 2012).

TDD é uma das práticas de desenvolvimento ágeis, nessa técnica é sugerida que o desenvolvedor escreva testes automatizados ao longo do desenvolvimento, antes mesmo da implementação em si. Com *TDD* tem-se um desenvolvimento incremental do código (SOMMERVILLE, 2011).

Uma das curiosidade sobre o *TDD*, é o fato de escrever um teste que falhe, para parte do código antes que este seja implementado. Esse conceito é conhecido como “*test-first development*” (GAŁEZOWSKI, 2018).

A prática “*Test-First*” significa passar os testes para o início do desenvolvimento, para que os testes automatizados possam ser escritos antes do código. Fazer essa mudança pode resultar em um *software* com maior qualidade e mais eficiência. A integração de desenvolvimento e testes evita erros desnecessários no final do ciclo de desenvolvimento, economizando tempo e aumentando a padronização para testes e garantia de qualidade (DUNNE, 2016).

A Figura 1, apresenta o ciclo de vida do *TDD*. O ciclo inicia com a criação de um teste que irá falhar, pois a funcionalidade ainda não foi desenvolvida (cor vermelha); Em seguida é escrito um código mínimo para fazer com que o teste avance um pouco (cor verde), então executa o teste novamente. O processo é executado até que o teste passe para o próximo passo; E por fim, faz a refatoração para melhoria do código escrito (cor roxa). Depois que todos testes forem executados com sucesso, é feita a implementação de uma próxima funcionalidade (PERCIVAL, 2017).

2.1.5.1 Vantagens e Desvantagens do *Test-Driven Development*

Segundo Sommerville (2011), ao empregar a técnica *TDD* é possível executar vários testes separados em poucos segundos. Uma das vantagens em usar o Desenvolvimento Orientado a Testes é elucidar as ideias do propósito de um segmento de código. Para escrever um teste é

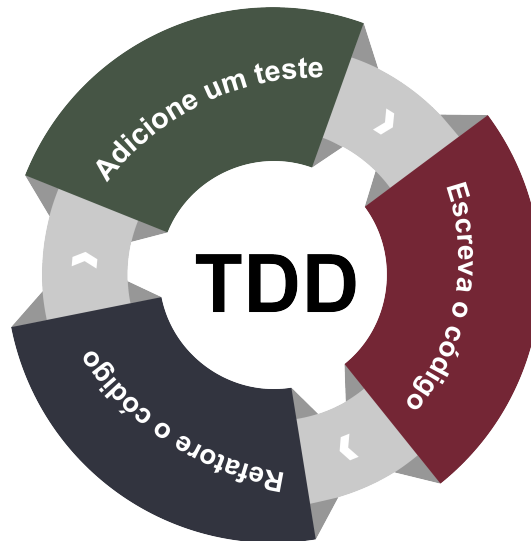


Figura 1 – Processo geral de *TDD*
Adaptado de (PICOLO, 2017)

necessário conhecimento prévio a respeito do problema que ele se destina resolver, com esse entendimento torna-se mais fácil escrever o devido código (SOMMERVILLE, 2011).

As equipes de desenvolvimento que adotam a técnica *TDD*, podem obter os seguintes benefícios (SOMMERVILLE, 2011):

- Cobertura de código - Todo segmento de código escrito deve ter pelo menos um teste associado. Portanto, cada código é testado enquanto está sendo escrito; assim, os defeitos são descobertos no início do processo de desenvolvimento.
- Teste de regressão - Um conjunto de testes é desenvolvido de forma incremental enquanto um programa é desenvolvido. Com *TDD*, os custos com o teste de regressão são reduzidos, uma vez que podem ser executados novamente de forma rápida e barata. Outra vantagem é que antes de adicionar uma nova funcionalidade, os testes existentes são capazes de revelar problemas caso essa mudança tenha introduzido novos *bugs* no sistema e se o novo código interage com o código existente conforme o esperado.
- Depuração simplificada - Quando um teste apresenta irregularidade, a localização do problema deve ser perceptível. O código recém-escrito precisa ser verificado e modificado imediatamente.
- Documentação de sistema - Os testes em si mesmos agem como um modelo de documentação, pois explicam o que o código deve estar fazendo. Ler os testes torna a compreensão do código mais fácil.

Diferentemente das vantagens apresentadas, também é possível analisar algumas desvantagens, tais como (SLYNGSTAD et al., 2008):

- *Design - TDD* geralmente contém pouco ou nenhum design do código, o que pode acarretar em ausência de documentação ao ocorrer algum defeito na aplicação, e então é notada a falta de informações que indique o estado anterior.
- Contexto - Deve existir um contexto para a escrita dos casos de teste e muitas vezes o tempo requerido para desenvolver o caso de teste pode tornar-se considerável. Além disso, não há como garantir que riscos relacionados com a falta de requisitos ou com requisitos erroneamente definidos serão eliminados.
- Refatoração - É extensamente empregado para atender a complexidade ao utilizar o *TDD*.
- Nível de habilidade - Necessita-se possuir um bom nível de experiência e conhecimento para desenvolver e conservar a precisão de teste em *TDD*.

Analisando as vantagens e desvantagens relacionadas quanto a utilização de *TDD* é possível notar que muito dos resultados serão bem aproveitados no longo prazo, pois inicialmente a curva de aprendizado requer muita disciplina para que a equipe não desista, acusando o *TDD* de redundante e exaustivo. Outro ponto a ser mencionado é que o *TDD* não assegurará o cumprimento do prazo de entrega estimado, porém propõe que com a realização dos testes é possível diminuir o tempo de correção do *software* (PERCIVAL, 2017).

2.2 DevOps

Este tópico apresenta uma visão geral sobre a cultura *DevOps* no ciclo de desenvolvimento de *software*, seus conceitos base, características e terminologias adotadas. Bem como, sua relação com qualidade de *software*.

2.2.1 O que é DevOps

A palavra "*DevOps*" foi usada pela primeira vez, por Debois (2010), que desde então é considerado o padrinho do movimento. Formada pela fusão de duas palavras: Desenvolvimento (Dev) e Operações (Ops). Trata-se de um movimento que está sendo construído em torno de pessoas que acreditam que a aplicação de tecnologia e atitudes apropriadas pode revolucionar o mundo do desenvolvimento e da entrega de *software* (DEBOIS, 2010).

Segundo AWS (2018), o movimento *DevOps* pode ser definido como a combinação entre práticas culturais e ferramentas, pois essa combinação aumenta a capacidade de uma empresa de disponibilizar aplicativos e serviços em alta velocidade, otimizando e aperfeiçoando produtos em um ritmo mais rápido, quando comparada às empresas que usam os processos tradicionais de desenvolvimento de *software* e gerenciamento de infraestrutura

A Figura 2, ilustra a combinação entre equipes, sendo o *DevOps* o resultado dessa integração. Para Sharma e Coyne (2017), o *DevOps* é uma abordagem que combina as equipes de

desenvolvedores, operações e equipe de controle de qualidade, visando melhorar a comunicação entre equipes de modo que a entrega das soluções sejam feitas com maior agilidade, reduzindo assim o tempo necessário para o *feedback* dos clientes (SHARMA; COYNE, 2017).

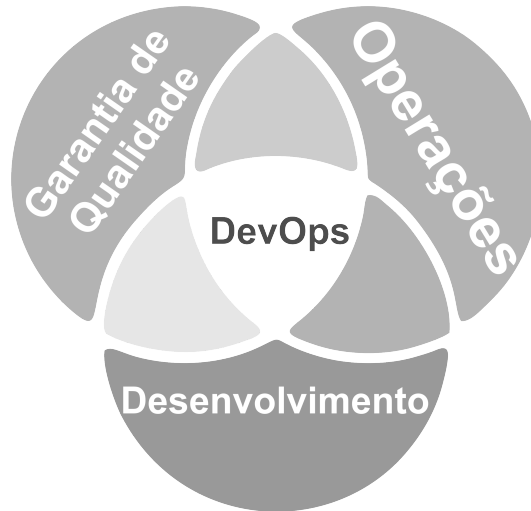


Figura 2 – Abstração do conceito *DevOps*
Fonte: Adaptado de (SADASIVAN, 2014).

O *DevOps* rompe as barreiras entre as equipes, enfatizando a comunicação, a colaboração e integração entre desenvolvedores de *software* (testadores, programadores, controle de qualidade) e profissionais de operações (administradores de sistema, técnicos de rede, DBAs) obtendo assim resultados melhores e mais rápidos durante o ciclo de desenvolvimento (SADASIVAN, 2014).

2.2.2 Pilares do *DevOps*

Os pilares do *DevOps* são conhecidos através do acrônimo "CAMS" (*Culture, Automation, Measurement and Sharing*), que foi baseado nas áreas codificadas por Debois (2012): cultura, automação, medição e compartilhamento, podemos ver na Figura 3 esses conceitos:

Durante a implantação do *DevOps* é necessário fazer melhorias na estrutura tradicional de uma organização, pois essa nova cultura deve abranger todos os envolvidos no *DevOps*.

O modelo CAMS foi popularizado por Willis e Edwards (2010), que apresentou inicialmente os conceitos dos pilares do *DevOps*. Essa é a base para mudar a estrutura tradicional de uma organização para uma cultura *DevOps* ou fazer melhorias em uma estrutura já existente (WILLIS; EDWARDS, 2010):

1. Cultura - Pessoas acima dos processos. O *software* é feito por e para pessoas. Se não tiver um cultura, todas as tentativas de automação serão ineficazes.
2. Automação - Começa quando se entende a cultura. Neste ponto, as ferramentas podem começar a unir seu conjunto de automação para o *DevOps*. Ferramentas para gerenciamento de versões, provisionamento, gerenciamento de configuração, integração de sistemas,



Figura 3 – Modelo CAMS - Pilares do *DevOps*
Adaptado de (SADASIVAN, 2014)

monitoramento e controle e orquestração tornam-se peças importantes na construção de uma estrutura *DevOps*.

3. Métricas - Se não for possível medir, não pode melhorar. Uma implementação bem-sucedida do *DevOps* medirá tudo o que puder com a maior frequência possível. Métricas de desempenho, métricas de processo e até métricas de pessoas.
4. Compartilhamento - O compartilhamento é o *loopback* (laço de retorno) no ciclo do CAMS. Criar uma cultura onde as pessoas compartilham ideias e problemas é fundamental, pois cria um *feedback* aberto auxiliando-as a melhorar.

As bases fundamentais para o sucesso do *DevOps*, segundo Hüttermann (2012), são uma cultura de confiança e um sentimento de companheirismo. Os benefícios de uma empresa que adota *DevOps* são: maior comunicação e colaboração entre áreas de desenvolvimento e infraestrutura, maior qualidade de *software*, maior frequência de entrega, redução de falhas, maior estabilidade e aumento do valor do negócio (HÜTTERMANN, 2012).

2.2.3 Terminologias *DevOps*

As atividades de *DevOps* são realizadas de forma unificada e constante conforme pode ser vista na Figura 4, onde observa-se que no ciclo de desenvolvimento são trabalhados os processos de Desenvolvimento Ágil, já no ciclo de Operações são realizadas as atividades de Infraestrutura Ágil (LEE, 2017).

No Desenvolvimento Ágil, engloba vários ciclos de planejamento seguindo (DIAS, 2018):

- Integração contínua - A CI tem por objetivo identificar e corrigir erros mais rapidamente, melhorar a qualidade do *software* e reduzir o tempo necessário para validar e lançar novas

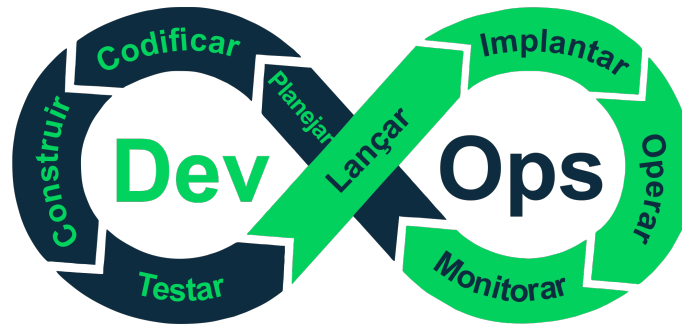


Figura 4 – Fluxo pipeline *DevOps*
Adaptado de (LEE, 2017)

atualizações de *software*. O código é testado e preparado diretamente (automação) antes de ser liberado para a produção.

- Entrega contínua - Conjunto de práticas cujo objetivo é garantir que alterações ou novas versões de *software*, sejam colocadas no ambiente de produção a qualquer momento.
- Implantação contínua - É uma extensão da integração contínua, com o objetivo de minimizar o tempo de espera em produção realizando *deploys* automáticos no processo.

Infraestrutura Ágil acompanha o desenvolvimento ágil e compreende (DIAS, 2018):

- Infraestrutura como Código - Tem por objetivo utilizar a abordagem do ciclo de desenvolvimento de *software* para realizar configuração da infraestrutura.
- Gerência de Configuração - Utilizam ferramentas que controlam a configuração das máquinas.
- Provisionamento - Prover os serviços e configurações de forma automatizada.

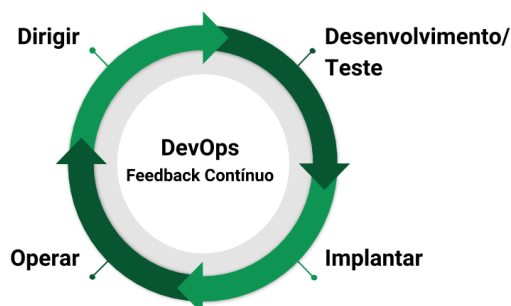


Figura 5 – Arquitetura de referência *DevOps*
Adaptado de (SHARMA; COYNE, 2017)

A Figura 5 propõe quatro conjuntos de passos a serem adotados para a iniciar a implantação do *DevOps* (SHARMA; COYNE, 2017), são eles:

- Dirigir - Este passo consiste em uma prática que se concentra em definir metas de negócios e ajustá-las com base no *feedback* do cliente: planejamento de negócios contínuo (*Continuous Business Planning*).
- Desenvolvimento/Teste - Este passo possui duas práticas: desenvolvimento colaborativo e testes contínuos (*Colaborative Development e Continuous Testing*). O desenvolvimento colaborativo permite que esses profissionais trabalhem em conjunto, proporcionando um conjunto comum de práticas e plataformas, que eles podem usar para criar e entregar *softwares*.

Um recurso central incluído no desenvolvimento colaborativo é integração contínua, popularizada pelo movimento ágil. A ideia é que os desenvolvedores integrem regularmente seu trabalho com o de outros membros de sua equipe e, em seguida, teste o trabalho integrado. No caso de sistemas complexos constituídos por vários sistemas ou serviços, os desenvolvedores também integram regularmente seu trabalho com outros sistemas e serviços.

Testes contínuos significam testes constantes no ciclo de vida, o que resulta em custos reduzidos, ciclos curtos de testes e *feedback* contínuo sobre a qualidade. Este processo enfatiza a integração das atividades de desenvolvimento e teste para garantir que a qualidade seja feita desde o início do ciclo e não algo deixado para depois. Isso é proporcionado pela utilização de recursos como testes automatizados e virtualização de serviços. A virtualização de serviços torna viável teste contínuo, pois simulam ambientes de produção.

- Implantar - O objetivo do lançamento e implantação contínuo é lançar novos recursos para clientes e usuários o mais breve possível. A maioria das ferramentas e processos que compõem o núcleo do *DevOps* possui tecnologias para facilitar a integração contínua (*Continuous Release and Deployment*).
- Operar - Esse passo inclui duas práticas: monitorar o desempenho das aplicações em produção (*Continuous Monitoring*) e receber *feedback* dos clientes (*Continuous Customer Feedback & Optimization*). O monitoramento contínuo fornece dados e métricas em diferentes fases do ciclo de entrega. Essas métricas permitem que as partes envolvidas reajam, melhorando ou alterando as características entregues.

Para que as organizações forneçam *software* continuamente, é necessário garantir que os testes e procedimentos relacionados à qualidade sejam projetados para se adequarem ao fluxo de desenvolvimento, integração e implantação. Os procedimentos e tecnologias de testes tradicionais não estão preparados para lidar com a demanda cada vez maior de testes rápidos e contínuos (SHIMEL, 2018).

2.2.4 Qualidade de *Software* no ambiente *DevOps*

A garantia de qualidade geralmente era considerada inimiga dos desenvolvedores, como resultado, as equipes de garantia de qualidade recebiam baixo orçamento e pouca atenção (ELIAV, 2018).

Entretanto, as empresas estão gradualmente investindo mais em habilidades e ferramentas de *QA*. Isso começou com a origem do *Agile* e *DevOps* - metodologias que se concentram em velocidade, eficiência e qualidade. Desde então, os líderes passaram a entender que grandes produtos não podem ser alcançados sem o adequado controle de qualidade (RILEY, 2014).

Os procedimentos e tecnologias de testes tradicionais simplesmente não estão preparados para lidar com a demanda cada vez maior de testes rápidos. Como resultado, as organizações são incapazes de alcançar uma entrega contínua (SHIMEL, 2018) .

No entanto, para atender às metas do *DevOps*, o *QA* deve prover o processo de desenvolvimento automatizado e ajudar as equipes a alcançar uma qualidade de *software* maior e um tempo de entrega mais rápido. De acordo com Geer (2016), os especialistas e equipes de *QA* têm várias funções e responsabilidades no *DevOps*, que estão relacionadas abaixo:

- O *QA* deve evitar defeitos de *software* o mais cedo possível durante o processo de desenvolvimento de *software*, em vez de localizar e relatar erros de *software* depois que o *DevOps* tiver produzido o *software*. Isso ajudará a garantir que os clientes tenham o menor número de defeitos possível.
- O departamento de *QA* assume a responsabilidade pela melhoria contínua do *software* e pelo monitoramento da qualidade do *software* durante todo o ciclo de vida do desenvolvimento.
- Todo membro da equipe de *QA* é responsável por identificar problemas não apenas no produto, mas também no processo, e recomendar alterações sempre que possível para melhorar a entrega contínua de *software* de qualidade.
- Os testes devem estar em código, para publicar uma nova versão toda semana, todo dia ou toda hora. Não há espaço para testes manuais - a *QA* deve desenvolver sistemas de automação que garantam que os padrões de qualidade sejam mantidos.
- Regras de automação: o controle de qualidade deve automatizar tudo que pode ser automatizado.
- Os testadores de *QA* são os defensores da qualidade, influenciando os processos de desenvolvimento e operações. Eles não apenas encontram *bugs*; eles procuram por qualquer oportunidade de melhorar a qualidade do *software* através do processo de desenvolvimento de *software*.

- O *QA* deve ir além do teste funcional. Muitas organizações têm processos maduros para automatizar testes funcionais; elas estão começando a aplicar essas práticas a outras áreas de teste, como testes de segurança e de estresse. Em particular, o teste de carga e o teste de estresse são disciplinas críticas para as organizações de *DevOps* que está evoluindo em alta velocidade.

Nesse contexto, as práticas de *QA* ajudam muito a melhorar a qualidade do *software*, especialmente quando as organizações aceitam prontamente o movimento *DevOps* para facilitar o desenvolvimento e as operações eficazes de *software*, mantendo alta qualidade com a experiência de usuário (RILEY, 2014).

2.3 *TDD* e *DevOps*

A utilização de técnicas de teste são essenciais para garantir a qualidade de um *software* e por isso o objetivo deste trabalho é integrar a técnica de *TDD* no contexto *DevOps*.

2.3.1 Relação entre *TDD* e *DevOps*

O uso de *TDD* nos movimentos de *Agile* e *DevOps* tem sido impactante. Para que as equipes *DevOps* entreguem *software* com qualidade e de forma contínua é necessário utilizar técnicas ágeis que o auxiliem (MACVITTIE, 2017).

A relação entre *DevOps* e *TDD* é que um é facilitador para o outro. A técnica de *TDD* prover testes automatizados em ambiente de integração fornecido por equipes que adotam as práticas de *DevOps*. Ao criar os testes no início do desenvolvimento, proposta do *TDD*, os desenvolvedores economizam tempo de trabalho e minimizam os custos para encontrar defeitos.

Em uma experiência feita em estudos acadêmicos, segundo Mezak (2017), sugere que o *TDD* aumenta o tempo de desenvolvimento em aproximadamente 15% e também que o custo total de *software* desenvolvido por *TDD* é menor do que o *software* que usa apenas o teste de integração.

Sendo assim, o desenvolvimento orientado a testes trabalha desde o início de seu ciclo com o propósito de fornecer a maior cobertura de código possível em curto prazo, de forma a ter testes contínuos em um ambiente de integração contínua fornecido pela equipe que adota a cultura *DevOps* (MACVITTIE, 2017).

2.3.2 Plataforma de Integração - Jenkins

De maneira a possibilitar a integração real entre as áreas de *TDD* e *DevOps*, sugere-se o uso da ferramenta Jenkins², que foi projetada para ser um servidor de automação de código aberto,

² Disponível em: <<https://jenkins.io/>>

nele é possível automatizar diversas tarefas do ciclo de desenvolvimento, relacionadas à criação, teste, distribuição ou implementação de *software* (GASKELL et al., 2018).

Boaglio (2016) apresenta brevemente o que é Jenkins e o que ele pode fazer de forma a melhorar e automatizar algumas tarefas.

"O Jenkins é um servidor de integração contínua *open source* e é feito em Java, que disponibiliza mais de 1.000 *plugins* para suportar construção (*build*) e testes de virtualmente qualquer tipo de projeto. Com ele, é possível automatizar uma série de procedimentos, como *deploy* em um servidor, rodar um conjunto de testes, rodar *scripts* em um banco de dados, atualizar uma aplicação *web*, e até delegar a execução para outros servidores." (BOAGLIO, 2016, p.5-6).

O Jenkins se destaca dentre outras ferramentas de Integração Contínua, por causa de seus recursos disponíveis e capacidade. Sendo o *software* de sua categoria mais amplamente utilizado. Algumas características do Jenkins (LESZKO, 2017):

- Linguagens de Programação: Suporta a maioria das linguagens de programação e *frameworks*. Pode ser usado comandos *shell*, possibilitando a instalação de qualquer *software*, permitindo assim a automação de quase todas as atividades pensadas;
- Plugins: Jenkins tem uma grande quantidade de *plugins* disponíveis, permite também a criação de *plugins* próprios personalizados de acordo com cada necessidade pessoal;
- Portabilidade: O Jenkins foi escrito em Java, sendo assim, multiplataforma. É fornecido em várias versões: imagem Docker, War e arquivos binários para Windows, Mac e Linux;
- Gerenciamento de Código-Fonte: Suporta a maioria dos gerenciadores de código, devido a sua vasta comunidade e *plugins* quase todas as ferramentas de versionamento são atendidas;
- Distribuído: O Jenkins possui um recurso interno para o modo *master/slave*, que possibilita sua execução de forma distribuída em vários nós localizados em máquinas diversas. É possível também usar um ambiente com diferentes sistemas operacionais instalados;
- Descomplicado: Seu processo de instalação e configuração é simples. Não é preciso configurar *software* extra e nem banco de dados. Pode ser configurado por meio de *scripts* Groovy, arquivo XML e por meio da interface gráfica;
- Orientado a código: O próprio Jenkins pode ser configurado usando arquivos XML ou *scripts* do Groovy, permitindo manter a configuração no repositório do código-fonte de forma a auxiliar na automação de sua configuração.

Integração Contínua é uma das práticas importantes do desenvolvimento ágil, através dela é possível automatizar tarefas e executar testes automatizados. A cada integração é possível verificar de forma mais rápida a detecção de erros.

Para exemplificar a proposta de integração das técnicas ágeis apresentadas, bem como avaliar o conjunto de testes a serem realizados, utilizou-se uma aplicação simples como prova conceito, que consiste em uma aplicação de lista de tarefas, baseado no guia de Percival (2017). O desenvolvimento e ferramentas utilizadas são especificados no capítulo 3.

3 Metodologia

De maneira a cumprir os objetivos deste projeto serão apresentadas as ferramentas utilizadas. O trabalho é desenvolvido a partir do uso de *softwares* livres, em virtude da vasta documentação fornecida pela comunidade técnica, além de oferecer possibilidade de ajustes para cenários específicos de utilização.

Com o intuito de exemplificar a proposta de integração das técnicas ágeis apresentadas, optou-se por desenvolver um cenário de hospedagem de sistemas *web* dentro de um conjunto específico de plataformas, que visa atender a necessidade de hospedagem de projetos que requerem ou fazem uso de aplicações *web*.

3.1 Tipo de Pesquisa

O tipo de pesquisa utilizada compõe-se em aplicada e estudo de caso. Quanto à natureza a pesquisa é aplicada, pois "objetiva gerar conhecimentos para aplicação prática dirigidos à solução de problemas específicos"(PRODANOV; FREITAS, 2013).

Quanto aos procedimentos adotados a pesquisa é estudo de caso, pois "envolve o estudo profundo e exaustivo de um ou poucos objetos de maneira que permita o seu amplo e detalhado conhecimento"(YIN, 2001). Gil (2008) complementa ainda que esses tipos de pesquisas estão mais voltadas para a aplicação imediata de conhecimentos em uma realidade circunstancial, relevando a composição de teorias.

3.2 Materiais Utilizados

Para o provimento do cenário de testes, bem como avaliar as ferramentas e soluções estudadas, utilizou-se os materiais detalhados na Tabela 2.

Tabela 2 – Recursos/Ferramentas Utilizados

DESCRIÇÃO	VERSÃO / MODELO
Servidor ESXi 6.5 ³	Intel(R) Xeon(R) CPU E5-2603 v4
Sistema Operacional Linux	Debian GNU/Linux 9.5.0 (64-bit) ⁴ 4GB RAM
Docker	18.06.1-ce
Jenkins	2.138.2
Git	2.17.1
Sublime Text ⁵	3.1.1
Python	3.6.6
SonarQube	7.1

3.3 Descrição do Cenário de Testes

As ferramentas que compõem o cenário são: Linguagem de programação Python⁶ com framework Django⁷, Git⁸, Jenkins, SonarQube⁹ e Docker¹⁰. A descrição de cada uma das ferramentas, é destacado a seguir:

- Python - Linguagem de programação que permite trabalhar rapidamente e integrar sistemas de forma mais eficaz. É uma linguagem livre e multiplataforma. Será utilizada do início ao fim dentro do ambiente de testes;
- Django - *Framework Web* Python que incentiva o rápido desenvolvimento, cuida de grande parte do desenvolvimento para a *Web*, ajuda a evitar erros comuns de segurança. É grátis e de código aberto.;
- Git - Sistema de Controle de Versões (CVS) distribuído gratuitamente e de código aberto, projetado para lidar com projetos pequenos a grandes sistemas, com velocidade e eficiência. Seu diferencial é o modelo de ramificação, que permite ter várias ramificações locais que podem ser totalmente independentes umas das outras. A criação, fusão e exclusão dessas linhas de desenvolvimento leva pouco tempo;
- Jenkins - Servidor de automação independente e de código aberto que pode ser usado para automatizar todos os tipos de tarefas relacionadas à criação, teste e distribuição ou implementação de *software*.
- SonarQube - Plataforma *open source*, responsável por fazer análise automática de código fonte, medindo assim a qualidade do código de uma aplicação, detectando vulnerabilidades, *code smell*, *bugs* e outras métricas de qualidade.
- Docker - Ferramenta responsável por empacotar aplicações ou ambientes completo, em áreas conhecida como contêineres. A partir disso o ambiente inteiro torna-se portátil para qualquer outro local que contenha Docker instalado. Sua utilização reduz o tempo de *deploy*, pois sua replicação dispensa ajustes de ambiente (MOUAT, 2016).

O desenvolvimento foi realizado utilizando a técnica *TDD*, tendo como principais testes envolvidos, os testes unitários, testes funcionais e testes integrados (de integração), usando o servidor de Integração Contínua (CI), Jenkins.

De maneira a ilustrar como ocorre os testes, detalha-se a seguir o cenário implementado. Na Figura 6 é possível observar as etapas necessárias que a aplicação *web* será submetida, e

³ Disponível em: <<https://www.vmware.com/br/download.html>>

⁴ Disponível em: <<http://debian.c3sl.ufpr.br/debian-cd/9.5.0/>>

⁵ Disponível em: <<https://www.sublimetext.com/3>>

⁶ Disponível em: <<https://www.python.org/>>

⁷ Disponível em: <<https://www.djangoproject.com/>>

⁸ Disponível em: <<https://git-scm.com/>>

por conseguinte, são apresentados quais conjuntos de testes podem ser aplicados. Tem-se ainda a representação de como as ferramentas trabalham em conjunto de forma a demonstrar como funciona a técnica de teste *TDD* no cenário de integração contínua utilizando Jenkins, que constitui um modelo de implantação referenciado pela cultura *DevOps*.

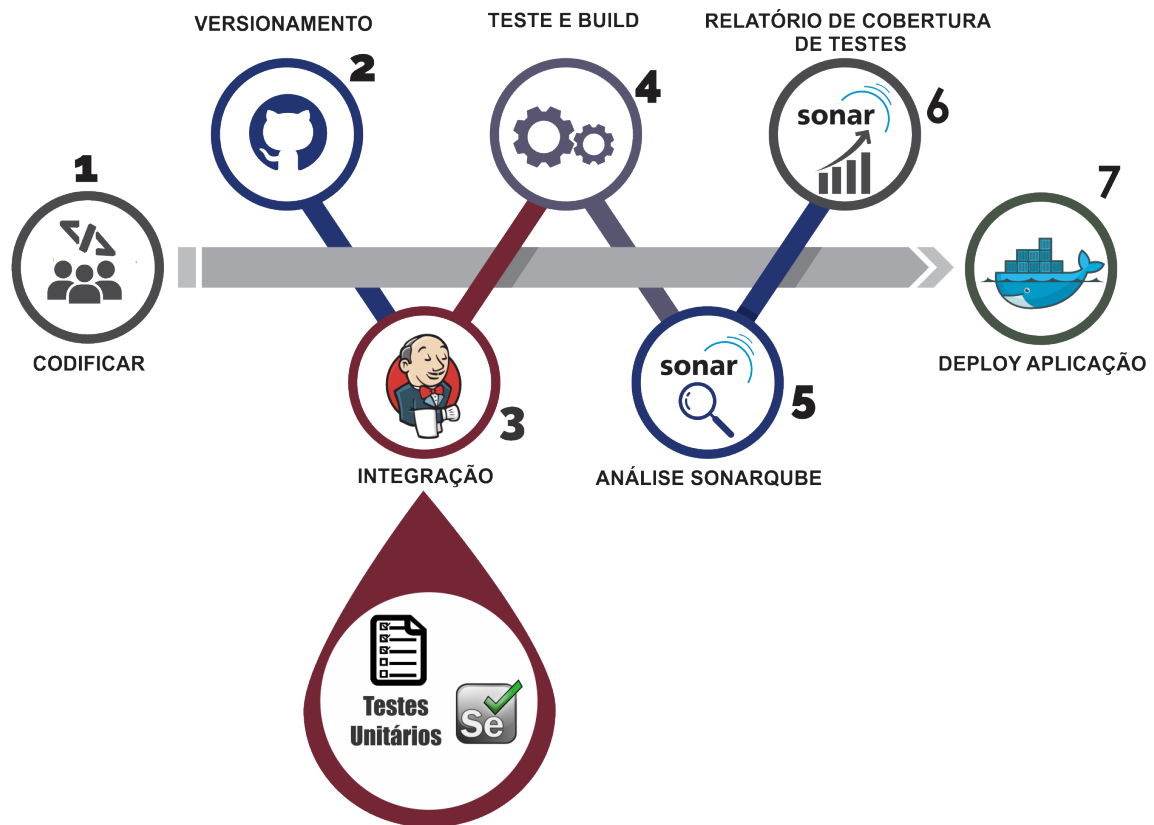


Figura 6 – Arquitetura da Plataforma de Hospedagem
Fonte: Autoria própria.

As etapas abaixo compreendem um modelo de como as técnicas podem trabalhar em conjunto, especifica-se a seguir o que cada uma significa:

- Etapa 1: A equipe/desenvolvedor inicia a implementação da aplicação *web*, usando a técnica *TDD*, elaborando primeiramente os testes;
- Etapa 2: Desde o início da codificação é importante manter o versionamento do código, neste caso, será utilizado o Github. Além de manter as versões armazenadas, caso surja algum problema, é possível fazer restauração para a versão anterior ao erro. Também é possível realizar o trabalho de maneira compartilhada entre a equipe, que é um dos princípios do *DevOps*;
- Etapa 3: Essa é uma das principais etapas, onde o *job* da aplicação é configurado no servidor de integração Jenkins. A URL do repositório é fornecida para fazer o *pull* do código. Configuração dos *plugins* necessários para análise e *deploy* da aplicação, informar também a frequência de execução do *job*. Configuração do *job* para caso ocorra alguma

falha no processo, o responsável faça imediatamente as correções. É possível receber a notificação via email.

- Etapa 4: Ao executar o *build* do *job*, o Jenkins faz a automatização dos testes, disparando de forma automática testes unitários e testes funcionais, garantindo que nada saia errado;
- Etapa 5: Após execução da etapa anterior, o escâner do SonarQube é acionado para iniciar a análise da qualidade do código, através da integração do SonarQube Plugin;
- Etapa 6: Após o processamento do escâner, é possível acessar o relatório de teste e a cobertura do código, fornecido pela plataforma do SonarQube. Algumas das informações reportadas são: código duplicado, padrões de codificação, erros potenciais, dentre outros;
- Etapa 7: Quando concluir todas essas etapas, a aplicação estará isolada, com o auxílio dos arquivos *Dockerfile* e *docker-compose* e disponível para ser feita o *deploy* no servidor de produção.

A ideia desse cenário é introduzir a técnica de *Test Driven Development* usando a linguagem de programação python. A aplicação *web* utilizada como prova de conceito, é bastante simples, visto que, o foco do trabalho é avaliar o que é a técnica do *TDD*, ou seja, como uma estrutura de *tests first* (sempre começar escrevendo os testes, e não a aplicação) pode guiar o processo de desenvolvimento; e, segundo, o que será analisado não é a complexidade da aplicação e sim a eficiência dos testes implementados.

3.3.1 Métricas

As métricas de teste são um padrão de medidas muito útil para a verificação da efetividade e da eficiência de diversas atividades do desenvolvimento de *software*. Para isso é necessário medir e manter o registro dessas medições, e, além disso, converter tais medidas em informações confiáveis e que tenham valor para os envolvidos (TRODÓ, 2009). Para analisar a efetividade dos testes do cenário, utilizou-se as métricas mostradas na Tabela 3 (TRODÓ, 2009).

Tabela 3 – Métricas de testes

MÉTRICAS	DESCRIÇÃO
Taxa de erros obtidos	Quantidade de ocorrências apresentando comportamento indevido.
Cobertura dos testes	Permite verificar se todas as funcionalidades do sistema estão sendo testadas, ou seja, se o teste cobre todas as funcionalidades.
Efetividade dos testes	O processo de teste é efetivo quando encontra defeitos.
Tempo do teste	Verifica o tempo gasto durante execução dos testes.
Índice de severidade de defeitos	É o índice que permite verificar se os defeitos são graves.

3.3.2 Breve modelagem da aplicação modelo

Para exemplificar a integração proposta, foi escolhido uma funcionalidade da aplicação *web* utilizada como modelo no processo de validação da arquitetura, que tem como objetivo criar listas de tarefas, similar ao exemplo usado no material de referência (PERCIVAL, 2017). A aplicação foi desenvolvida com base no *framework* Django, que utiliza o padrão MTV (*Model-Template-View*) (DJANGO, 2018), semelhante ao padrão de arquitetura MVC (*Model - View - Controller*) (PRESSMAN, 2011), conforme ilustrado na Figura 7.

A partir da análise da Figura 7, observa-se que no *framework* Django é usado o termo *Templates* para *Views*, ou seja, onde contém os códigos *HTMLs* da aplicação, responsável pela interface de interação usuário e aplicação. Já os *Controller* do padrão de arquitetura MVC, no *Django* é usado o termo *View* onde ficará as funções implementadas em Python que processam as entradas na interface. No pacote *Model* fica as classes modelos, responsável pela definição dos dados de entrada.

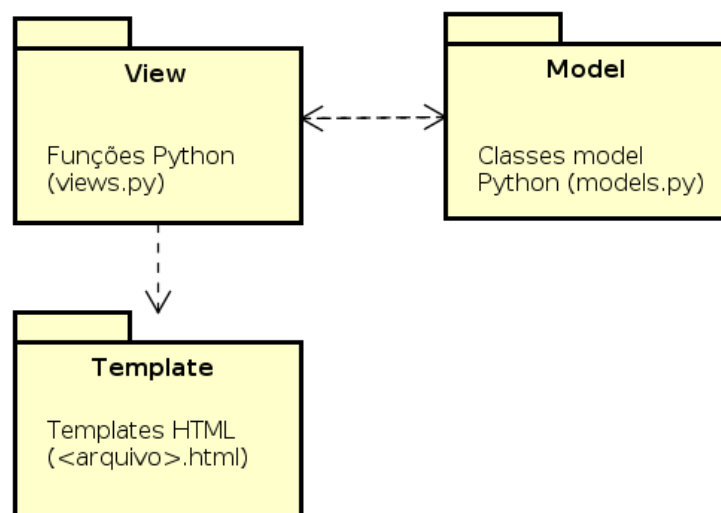


Figura 7 – Diagrama de Pacotes – UML

Fonte: Autoria própria.

Para a demonstração do funcionamento dos requisitos definidos da aplicação, utilizou-se recursos da linguagem *UML* (*Unified Modeling Language*). Na Figura 8 apresenta-se o diagrama de casos de uso, que ilustra o funcionamento geral da aplicação no contexto da arquitetura proposta.

De maneira a permitir um detalhamento maior, outros diagramas UML foram utilizados. Na Figura 9 demonstra o fluxo de atividades disponíveis para o usuário, bem como alguns procedimentos realizado pela aplicação.

⁹ Disponível em: <<https://www.sonarqube.org/>>

¹⁰ Disponível em: <<https://www.docker.com/l/>>

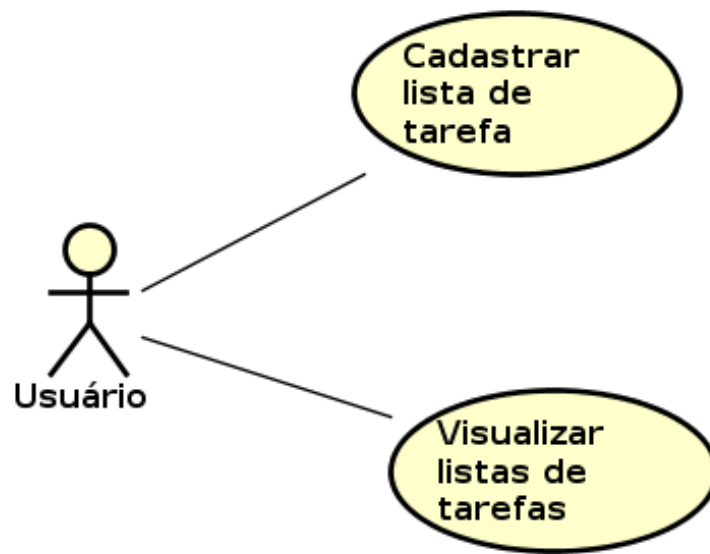


Figura 8 – Diagrama de Caso de Uso - UML
Fonte: Autoria própria.

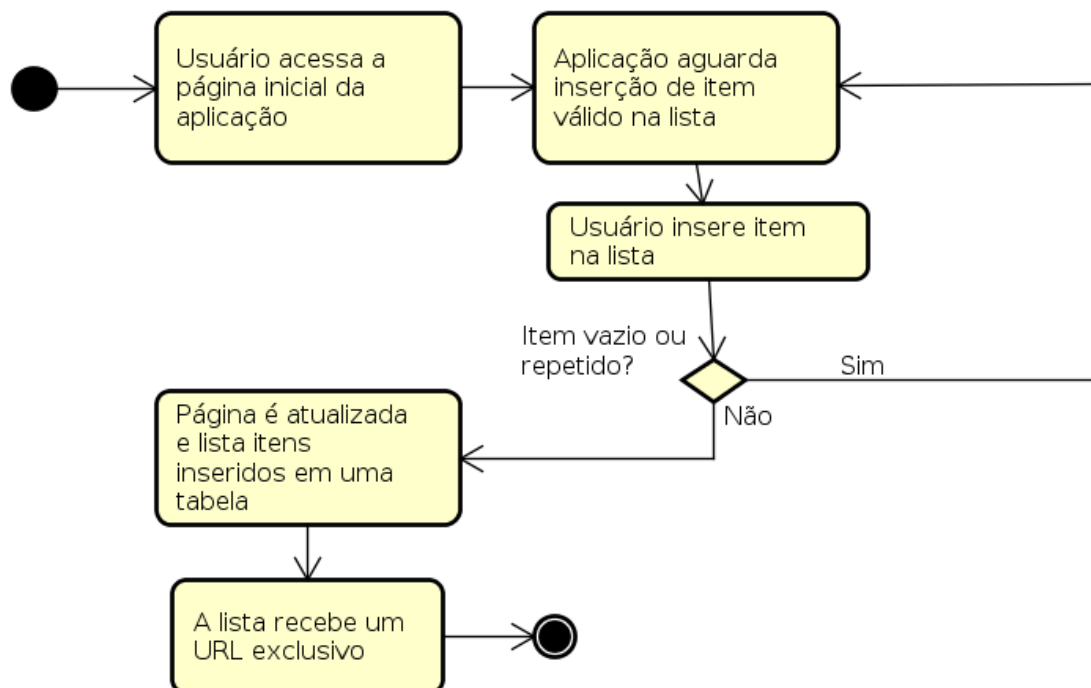


Figura 9 – Diagrama de Atividades - UML
Fonte: Autoria própria.

O diagrama de classes implementado no desenvolvimento da aplicação é ilustrado na Figura 10. Dessa forma é possível verificar como estão dispostos os pacotes, e como se dá a interação entre as classes. Para a confecção dos diagramas utilizou-se o *software* Astah UML¹¹, versão 8.0.

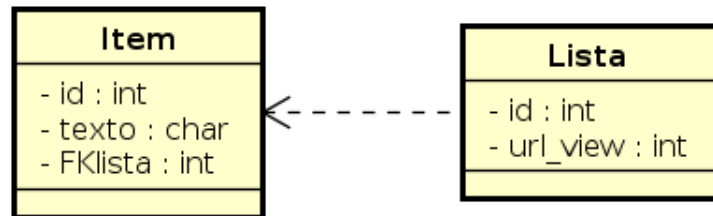


Figura 10 – Diagrama de Classes - UML
Fonte: Autoria própria.

3.3.3 Pseudocódigos dos estágios configurados

De maneira a simular o ambiente de hospedagem de sistemas *web*, utilizou-se quatro principais estágios de configuração no Jenkins, que atendem as necessidades do ciclo de teste, detalhados a seguir:

1. Integração com Git: Este *Job* tem como objetivo obter o código do repositório, instalar todas as dependências. A cada *commit* efetuado é realizado a execução de todo o pipeline configurado.
2. Integração dos Testes Unitários e Funcionais: Todos os testes unitários e funcionais são executados. Este estágio é executado periodicamente, sempre que ocorre algum novo *commit*.
3. Qualidade de código: Este *Job* permite gerar o relatório de cobertura de testes, verificação da existência de bugs, dentre outras informações.
4. *Deploy* da aplicação no Docker: Este *Job* permite a geração da imagem da aplicação e sua execução para o ambiente de testes, ou até mesmo produção.

No Jenkins um *Job* é qualquer projeto ou tarefa que se deseja executar e *build* é cada

¹¹ Disponível em: <<http://astah.net/editions/uml-new>>

uma das execuções do *Job*.

Algoritmo 1: Estágio de integração com Git

Entrada: Novo estágio

Saída: Relatório de sucesso ou falha

```
1 início
2   Configura o job/script para fazer o download do código do repositório
3   para este estágio faça
4       Verificar URL do repositório do Github;
5       Verificar CVS e construir periodicamente;
6       Construir workspace no Jenkins;
7       Clonar repositório remoto;
8       Buscar alterações no repositório;
9       Criar ambiente virtual do Python;
10      Instalar as dependências.
11  fim
12 fim
13 retorna Relatório sobre o sucesso ou falha da compilação
```

As entradas do Algoritmo 1 refere-se a configuração que o desenvolvedor deve fazer para atingir o *script* de integração com Github, podendo ser configurado via interface gráfica ou através do *Jenkinsfile*. A saída do processamento do algoritmo será uma mensagem reportada sobre o sucesso ou falha da compilação.

Algoritmo 2: ESTÁGIO DE INTEGRAÇÃO TESTES DE UNIDADE E FUNCIONAIS

Entrada: Novo estágio

Saída: Relatório de sucesso ou falha

```
1 início
2   Configura o job/script e construir o pipeline
3   para este estágio faça
4       Escrever testes unitários para o código;
5       Escrever testes funcionais para o código;
6       Fazer push para o repositório;
7       Disparar execução dos testes unitários;
8       Disparar execução dos testes funcionais.
9   fim
10 fim
11 retorna Status da execução dos testes
```

As entradas do Algoritmo 2, refere-se a configuração que o desenvolvedor deve fazer para atingir o estágio de Testes da aplicação, onde será verificado se o código faz o que promete. A saída do processamento do algoritmo será um relatório mostrando mensagem de sucesso ou

falha. Através desse *job*, as métricas que serão atendidas: taxa de erros obtidos, efetividade dos testes, tempo do teste.

Algoritmo 3: ESTÁGIO DE QUALIDADE DE CÓDIGO

Entrada: Novo estágio

Saída: Relatório de Análise Sonar

```

1 início
2   Executar o plugin SonarQube Scanner
3   para este estágio faça
4     Preparar SonarQube Scanner;
5     Informar chave do projeto;
6     Informar caminho do código no workspace;
7     Fornecer URL do servidor SonarQube, em que será feito a análise da aplicação;
8     Informar login para permitir acesso ao servidor SonarQube.
9   fim
10 fim
11 retorna Relatório Sonar de análise do código

```

As entradas do Algoritmo 3 refere-se a configuração que o desenvolvedor deve fazer para atingir o estágio de Qualidade de código. Para esse estágio é configurado todos os critérios descritos no algoritmo para integração do Sonar com o Jenkins.

Após feita os devidos ajustes, é retornado um relatório SonarQube com o resultado da execução, onde é exibido informações de qualidade do código, tais como as vulnerabilidades, "*code smells*" (más práticas de codificação), duplicidade de código, dentre outras. Através desse *job*, as métricas atendidas são: cobertura dos testes, onde será possível verificar se os testes cobre todas as funcionalidades e efetividade dos testes.

Algoritmo 4: ESTÁGIO DE *Deploy* DA APLICAÇÃO COM DOCKER

Entrada: Novo estágio

Saída: Construção da imagem Docker e *Build* da Aplicação

```

1 início
2   Executar o script Dockerfile
3   para este estágio faça
4     Criar Dockerfile, para construir imagem da aplicação;
5     Executar imagem para criar contêiner da aplicação.
6   fim
7 fim
8 retorna Deploy da aplicação no Docker

```

As entradas do Algoritmo 4, refere-se a configuração que o desenvolvedor deve fazer para atingir o *Deploy* da aplicação no contêiner Docker.

O arquivo *Dockerfile* possui todos os critérios para construção da imagem da aplicação, após gerar será possível utilizar por todos os membros de uma equipe de desenvolvedores, permitindo assim um ambiente padrão tanto na etapa de testes, homologação e produção. A execução da imagem gera o contêiner onde será possível acessar a aplicação.

4 Resultados

Neste tópico, serão mostrados os resultados obtidos a partir do conjunto de teste definido no capítulo 3. Os dados encontram-se organizados de forma cronológica de acordo com a execução de cada etapa. Alguns dados serão mostrados em tabelas e gráficos, seguidos de uma breve análise. Para tratar os dados, utilizou-se os recursos dos *plugins* configurados no Jenkins.

A seguir, os resultados apresentam-se através da arquitetura de integração contínua, conforme o detalhamento minucioso no capítulo 3.

Um dos objetivos do teste contínuo, é obter *feedback* o mais rápido possível, de forma que isso seja sinalizado se ocorrer algum erro. Para que isso ocorra é importante manter o ambiente de integração contínua bem configurado para que os erros sejam priorizados imediatamente.

4.1 Resultados do *Pipeline* de Integração Contínua

O Jenkins permite duas formas de configurar seu ambiente de integração contínua, a primeira é através da configuração de *Jobs Freestyle* para realizar as tarefas sequenciais. A outra maneira de configurar é com base no "*pipeline* como código".

A definição de um *Pipeline* é feito em arquivo de *script*, chamado por padrão de *Jenkinsfile*, ele fica junto com o projeto de origem no repositório de versionamento; tratando o *pipeline* de IC uma parte do aplicativo a ser versionado e revisado como qualquer outro código. Todas as etapas para criar uma aplicação, testá-la e, em seguida, entregá-la, estão configurada no *script*. Na Figura 11 é possível visualizar as etapas do *pipeline* configurado para o ambiente proposto.

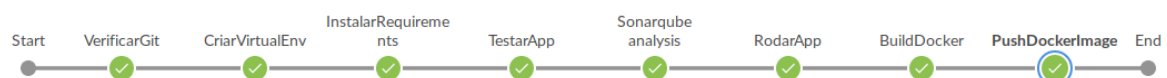


Figura 11 – *Pipeline* Jenkins
Fonte: Autoria própria.

Quando a execução de uma das etapas falha, o processamento é interrompido e nenhuma outra etapa seguinte será executada. Isso ocorre para que seja possível visualizar o ponto de falha e então corrigir.

Para atingir o objetivo do projeto, realizou-se testes tanto com a configuração via *Jobs Freestyle* como configuração via *script Jenkinsfile*.

4.1.1 Integração com git

As primeiras fases do *pipeline* funciona da seguinte maneira. Um desenvolvedor modifica o código no repositório, o servidor de Integração Contínua detecta a alteração e a construção é iniciada. Esse estágio contém três etapas:

1. Verificar Git: Este estágio faz o download do repositório e compila o código fonte;
2. Criar Virtual Env : Este estágio criar o ambiente virtual utilizado pelo Python;
3. Instalar *Requirements* : Este estágio executa a instalação de todas as dependências para que a aplicação rode no ambiente virtual do estágio anterior. Todas as dependências estão salvas no repositório com o nome padrão "*requirements.txt*".

4.1.2 Integração dos testes unitários

Para a implementação dos testes de unidade, foi usado o módulo padrão do Django: *unittest*, onde os testes são definidos baseados em classes Python. Os testes foram criados seguindo as fases vermelha, verde e refatoração, de acordo com a literatura *TDD* (PERCIVAL, 2017).

Para exemplificar essa integração realizou-se os seguintes testes unitários, conforme Tabela 4, sendo ao todo 34 testes unitários escritos, os casos de testes podem ser identificados na primeira coluna, cada teste está separado por pacote conforme mencionado na Figura 7, tendo um conjunto de testes *paramodels*, *views* e *forms* que podem ser conferidos na segunda coluna. As linhas da terceira coluna, refere-se sobre a localização inicial do teste dentro do arquivo, a quarta coluna é o nome do teste em si e por fim, a quinta coluna apresenta o tempo de execução do teste em segundos (s).

Tabela 4 – Casos de teste e Testes unitários (PERCIVAL, 2017)

Casos de teste (<i>TestCase</i>)	Arquivo	Linhas	Nome do teste	Tempo em s
lists.tests.test_forms.ItemFormTest	lists/tests/test_forms.py	6	test_form_item_input_has_placeholder_and_css_classes	0.467534780502
lists.tests.test_forms.ItemFormTest	lists/tests/test_forms.py	16	test_form_save_handles_saving_to_a_list	0.00456261634826
lists.tests.test_forms.ItemFormTest	lists/tests/test_forms.py	11	test_form_validation_for_blank_items	0.00370121002197
lists.tests.test_forms.ExistingListItemFormTest	lists/tests/test_forms.py	27	test_form_renders_item_text_input	0.0067799091339
lists.tests.test_forms.ExistingListItemFormTest	lists/tests/test_forms.py	48	test_form_save	0.0070147514343
lists.tests.test_forms.ExistingListItemFormTest	lists/tests/test_forms.py	33	test_form_validation_for_blank_items	0.0074317455291
lists.tests.test_forms.ExistingListItemFormTest	lists/tests/test_forms.py	40	test_form_validation_for_duplicate_items	0.011132717132
lists.tests.test_models.ItemModelTest	lists/tests/test_models.py	36	test_CAN_save_same_item_to_different_lists	0.0100643634796
lists.tests.test_models.ItemModelTest	lists/tests/test_models.py	20	test_cannot_save_empty_list_items	0.0087070465087
lists.tests.test_models.ItemModelTest	lists/tests/test_models.py	8	test_default_text	0.00260233879089
lists.tests.test_models.ItemModelTest	lists/tests/test_models.py	28	test_duplicate_items_are_invalid	0.0052752494812
lists.tests.test_models.ItemModelTest	lists/tests/test_models.py	12	test_item_is_related_to_list	0.0078318119049
lists.tests.test_models.ItemModelTest	lists/tests/test_models.py	45	test_list_ordering	0.00431847572326
lists.tests.test_models.ItemModelTest	lists/tests/test_models.py	56	test_string_representation	0.0033023357391
lists.tests.test_models.ListModelTest	lists/tests/test_models.py	62	test_get_absolute_url	0.0198967456817
lists.tests.test_views.HomePageViewTest	lists/tests/test_views.py	14	test_home_page_uses_item_form	0.02795433998
lists.tests.test_views.HomePageViewTest	lists/tests/test_views.py	10	test_uses_home_template	0.0069494247436
lists.tests.test_views.NewListTest	lists/tests/test_views.py	20	test_can_save_a_POST_request	0.0073890686035
lists.tests.test_views.NewListTest	lists/tests/test_views.py	42	test_for_invalid_input_passes_form_to_template	0.0090663433074
lists.tests.test_views.NewListTest	lists/tests/test_views.py	32	test_for_invalid_input_renders_home_template	0.0087761878967
lists.tests.test_views.NewListTest	lists/tests/test_views.py	46	test_invalid_list_items_arent_saved	0.0089662075042
lists.tests.test_views.NewListTest	lists/tests/test_views.py	27	test_redirects_after_POST	0.0221126079559
lists.tests.test_views.NewListTest	lists/tests/test_views.py	37	test_validation_errors_are_shown_on_home_page	0.00725579261779
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	106	test_POST_redirects_to_list_view	0.0191686153411
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	93	test_can_save_a_POST_request_to_an_existing_list	0.0086915493011
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	72	test_displays_item_form	0.0109796524047
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	78	test_displays_only_items_for_that_list	0.0140523910522
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	140	test_duplicate_item_validation_errors_end_up_on_lists_page	0.0148992538452
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	123	test_for_invalid_input_nothing_saved_to_db	0.0133643150329
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	132	test_for_invalid_input_passes_form_to_template	0.0137376785278
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	127	test_for_invalid_input_renders_list_template	0.0145280361175
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	136	test_for_invalid_input_shows_error_on_page	0.0127415657043
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	66	test_passes_correct_list_to_template	0.0091776847839
lists.tests.test_views.ListViewTest	lists/tests/test_views.py	61	test_uses_list_template	0.024548530578

Tabela 5 – Casos de teste e Testes funcionais (PERCIVAL, 2017)

Casos de teste (<i>TestCase</i>)	Arquivo	Linhas	Nome do teste	Tempo em s
functional_tests. test_list_item_validation. ItemValidationTest	functional_tests/test_list_item_validation.py	45	test_cannot_add_duplicate_items	0.001684904098510
functional_tests. test_list_item_validation. ItemValidationTest	functional_tests/test_list_item_validation.py	7	test_cannot_add_empty_list_items	0.001163959503173
functional_tests. test_simple_list_creation. NewVisitorTest	functional_tests/test_simple_list_creation.py	10	test_case_start_a_list_for_one_user	0.001213788986206
functional_tests. test_simple_list_creation. NewVisitorTest	functional_tests/test_simple_list_creation.py	47	test_multiple_users_can_start_lists_at_different_urls	0.001142024993896

Esse estágio da integração executa o conjunto de testes unitários descritos na Tabela 4.

Ao fazer o comando de execução dos testes unitários escritos em Django todos os testes passam com sucesso, conforme apresentado no *log* de saída abaixo:

```
+ python manage.py test lists
.....
-----

Ran 34 tests in 0.295s

OK
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Para gerar o gráfico com os resultado dos testes unitários foi usado o plugin *Test Results Analyzer*¹², que gera um gráfico com o resultados dos testes a partir de um arquivo XML. Para obter o arquivo XML, necessitou-se alterar o comando de execução dos testes, e não houve alteração da saída, executando todos os testes com sucesso, como o esperado:

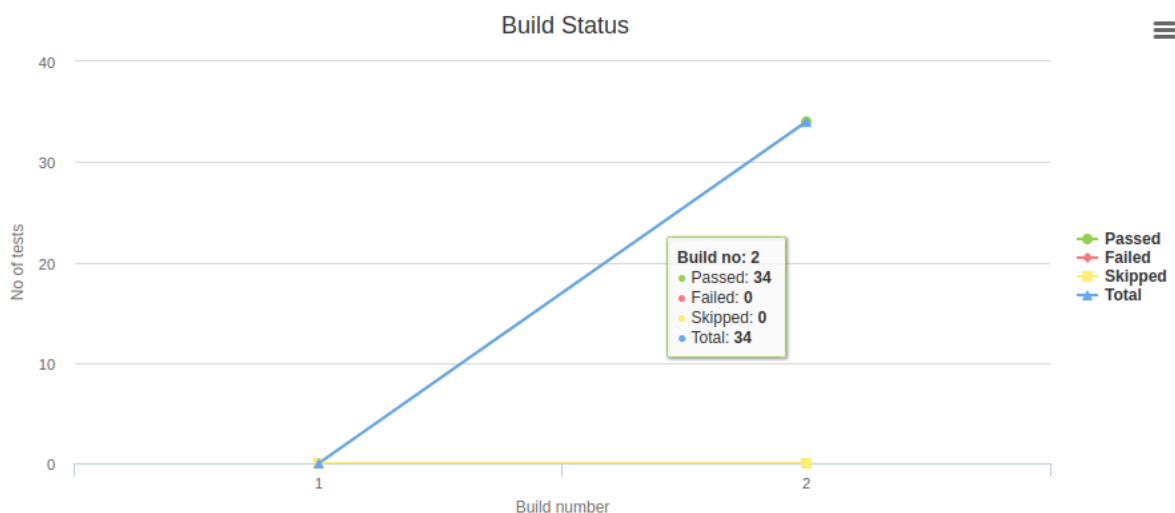


Figura 12 – Gráfico de linha com o resultado dos testes unitários
 Fonte: Autoria própria.

Verifica-se no gráfico da Figura 12 o resultado da execução de todos os testes unitários, total de 34 testes que passaram. É apresentado também os números dos *builds* em que foi executado o comando, que neste caso foram 2 *builds*.

¹² Disponível em <<https://wiki.jenkins.io/display/JENKINS/Test+Results+Analyzer+Plugin>>

4.1.3 Integração dos testes funcionais

Para a implementação dos testes funcionais, utilizou-se o *framework* Selenium¹³, que permite realizar os testes de uma aplicação de forma automatizada. Também fez-se uso das fases do *TDD* (vermelha, verde e refatoração) para sua escrita. Para demonstrar os testes das funcionalidades do sistema realizou-se os seguintes testes funcionais, conforme Tabela 5, sendo ao todo 4 testes funcionais.

A partir da análise da tabela nota-se na primeira coluna todos os casos de testes para as funcionalidades do sistema *web*. Os testes funcionais separou-se por funcionalidade do sistema, que verificam as atividades do usuário, conforme mencionado na Figura 9. As linhas da terceira coluna da Tabela 5, refere-se sobre a localização inicial do teste dentro do arquivo, a quarta coluna é o nome do teste em si e por fim, a quinta coluna da tabela apresenta o tempo de execução do teste em segundos (s).

Ao executar o comando dos testes funcionais todos os testes passam com sucesso, tendo como tempo de duração 49.632 segundos, conforme apresentado no *log* de saída abaixo:

```
+ python manage.py test functional_tests
```

```
....
```

```
-----  
Ran 4 tests in 49.632s
```

```
OK
```

```
Creating test database for alias 'default'...
```

```
System check identified no issues (0 silenced).
```

```
Destroying test database for alias 'default'...
```

Os testes funcionais rodam diretamente em em um navegador com auxílio da ferramenta Selenium, simulando exatamente como o usuário faria. No caso da integração adotou-se o Plugin *Xvfb*¹⁴, que executa todas as operações gráficas na memória virtual sem mostrar nenhuma saída de tela, apenas a mensagem de sucesso ou erro, conforme *log* mencionado anteriormente.

Para gerar o gráfico do resultado dos testes funcionais no Jenkins, necessitou-se alterar o comando de execução dos testes, porém a saída não foi como o esperado, uma vez que o *Plugin Test Results Analyzer*, não conseguiu gerar um gráfico confiável. O insucesso talvez tenha sido a adaptação necessária do comando para gerar o XML, visto que não foi possível a integração entre os plugins *Xvfb* e *Test Results Analyzer*.

A Figura 13 demonstra que a execução dos quatro testes falham. Para que as etapas seguintes continuassem o comando anterior ao erro, foi reescrito. A retirada do comando para gerar o gráfico não interfere no progresso do *pipeline*, uma vez que os testes funcionais rodaram

¹³ Disponível em: <<https://www.seleniumhq.org/>>

¹⁴ Disponível em <<https://wiki.jenkins.io/display/JENKINS/Xvfb+Plugin>>

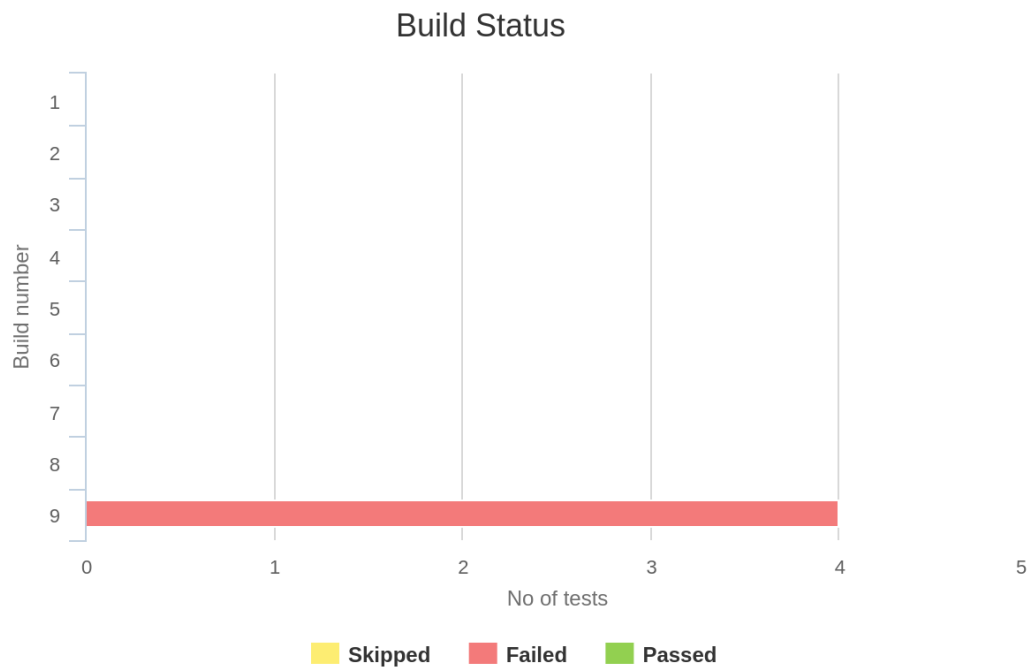


Figura 13 – Gráfico de barra com o resultado dos testes funcionais
Fonte: Autoria própria.

com sucesso com o auxílio do *plugin Xvfb*.

4.1.4 Integração com SonarQube

A integração do Jenkins com SonarQube deu-se através do SonarQube Plugin¹⁵, ao executar o *job* de integração do SonarQube obteve-se um relatório com as informações que podem ser observadas na Figura 14.



Figura 14 – Relatório geral da análise do SonarQube

Fonte: Autoria própria.

Os resultados são obtidos através da análise do código fonte. A geração dos painéis de controle de qualidade apontam para as características a seguir:

- **Bugs:** Foi encontrado 14 *bugs* no projeto analisado, o código está propenso a gerar comportamento inesperado. A classificação de confiabilidade é *D* (Quando existe pelo menos um *bug* crítico), todos os critérios para classificação pode ser visto na Tabela 6.

A Tabela 7 extraída a partir da análise realizada pelo SonarQube detalha quais tipos de *bugs* foram encontrados, indicando qual comportamento aguardado e sugerindo um tempo de esforço para fazer a correção. O tempo varia entre 5 a 30 minutos dependendo da atividade.

É possível ainda, saber a gravidade de cada um dos *bugs* encontrados, nesse caso temos 5 com grau crítico e 9 com grau maior. O status de todos estão aberto, ou seja, ainda não foram solucionados, outros status que podem ser substituído são: reaberto, confirmado, resolvido e fechado.

- **Vulnerabilities (Vulnerabilidades):** Há uma vulnerabilidade no código, o que pode tornar a segurança exposta a invasões. A vulnerabilidade encontrada pode ser visualizada na Tabela 8. A insegurança encontrada trata de uma configuração requerida com relação ao endereço IP, onde sugere que o mesmo seja configurável.

A gravidade dessa vulnerabilidade é menor. E sua classificação de segurança é *B* (Quando o problema de vulnerabilidade é do tipo Menor), os demais critérios para classificação pode ser visto na Tabela 6. O *status* encontra-se aberto, aguardando ser resolvido e o tempo sugerido para realizar essa atividade é de 30 minutos.

- **Code Smells (Cheiro de Código):** *Code smell* é quando ocorre más práticas de codificação e são medidos principalmente em termos do tempo que levarão para serem corrigidos. Na

¹⁵ Disponível em <<https://wiki.jenkins.io/display/JENKINS/SonarQube+plugin>>

Tabela 9 apresenta os 14 *Code Smell* detectados e a ação que deve ser feita para corrigi-los. A classificação de sustentabilidade recebida foi A (Quando o tempo de esforço para consertar todos os *code smells* é menor igual a 5%), os demais critérios para classificação pode ser visto na Tabela 6.

Ao lado de cada *code smell* tem a sugestão do tempo que levará para ser corrigido, variando de 2 a 20 minutos, a depender da atividade. A criticidade, nesse caso são de 4 *code smells* com grau menor e 10 com grau maior. Todos aguardam solução e por isso estão com status aberto, que também podem ser alterados para uma dessas opções: reaberto, confirmado, resolvido e fechado.

- *Duplications* (Duplicidade): Porcentagem de códigos duplicados, essa é uma das partes mais críticas da análise obtida, pois as cores classificam onde deve ser concentrado os esforços para correção, conforme Figura 14 o mais crítico está sendo os 43.9% de duplicidade de código.

A Tabela 10 demonstra a relação de arquivos com duplicidade, a porcentagem e quantidade de linhas duplicadas por arquivo. Os dois arquivos que mais contém linhas duplicadas são os arquivos estáticos *bootstrap.js* com 99.4% de linhas duplicadas e o arquivo *bootstrap.bundle.js* com 61% de linhas duplicadas, ambos são arquivos de *javascript* adicionado ao projeto em uma fase de estilização não concluída.

É possível notar que os códigos da aplicação desenvolvida não geraram duplicidades. Outros campos reportados foram a quantidade de arquivos envolvidos em duplicações e número de blocos duplicados de linhas, tendo valor apenas os arquivos *javascript*. A avaliação recebida por todos foi A, que é quando o índice de erros técnicos é inferior a 5% (taxa definida pelo SonarQube).

- *Coverage* (Cobertura): Porcentagem de cobertura do código-fonte pelos testes de unidade. A Tabela 11 apresenta a relação de arquivos que não foram cobertos pelos testes unitários, segundo análise do SonarQube. É demonstrado o número de linhas de código e o número de condições, que não são cobertas por testes unitários.

Apesar desse resultado mostrando ausência de cobertura do código por testes de unidade, o projeto em sua maioria foi sim efetuado a escrita dos testes unitário com a biblioteca *unittest* do Django, porém na análise feita pelo *scanner* do SonarQube ele não conseguiu fazer conexão com esses testes, exibindo um percentual de 0%.

Tabela 6 – Definição das Métricas do SonarQube (S.A, 2018)

Classificação de Segurança	Classificação de Confiabilidade	Classificação de Sustentabilidade
A = 0 Vulnerabilidades	A = 0 Bugs	<= 5% do tempo que já passou para o aplicativo, a classificação é A
B = pelo menos 1 Vulnerabilidade Menor	B = pelo menos 1 Bug Mínimo	entre 6 a 10% a classificação é um B
C = pelo menos 1 Vulnerabilidade Maior	C = pelo menos 1 Bug Principal	entre 11 a 20% a classificação é um C
D = pelo menos 1 Vulnerabilidade Crítica	D = pelo menos 1 Bug Crítico	entre 21 a 50% a classificação é um D
E = pelo menos 1 Vulnerabilidade do Bloqueador	E = pelo menos 1 Bug do Bloqueador	qualquer valor acima de 50% é um E

Tabela 7 – Relatório de *Bugs* SonarQube

Bug	Tipo	Tempo de correção	Gravidade	Status
1	Esta função declarada na linha 3938 espera 1 argumento, mas 2 foram fornecidos.	10min	Crítico	Aberto
2	Esta função declarada na linha 4997 espera 1 argumento, mas 2 foram fornecidos.	10min	Crítico	Aberto
3	Esta função declarada na linha 1421 espera 1 argumento, mas 2 foram fornecidos.	10min	Crítico	Aberto
4	Esta função declarada na linha 2480 espera 1 argumento, mas 2 foram fornecidos.	10min	Crítico	Aberto
5	Altere essa condição para que ela não seja sempre avaliada como "false"; algum código subsequente nunca é executado.	15min	Maior	Aberto
6	TypeError pode ser lançado como "a" pode ser nulo ou indefinido aqui.	10min	Maior	Aberto
7	TypeError pode ser lançado como "b" pode ser nulo ou indefinido aqui.	10min	Maior	Aberto
8	Mova essa operação de "sort" da matriz para uma instrução separada.	5min	Maior	Aberto
9	Mova essa operação de "sort" da matriz para uma instrução separada.	5min	Maior	Aberto
10	TypeError pode ser lançado como "parser" pode ser nulo ou indefinido aqui.	10min	Maior	Aberto
11	Altere essa condição para que ela não seja sempre avaliada como "false"; algum código subsequente nunca é executado.	15min	Maior	Aberto
12	Altere essa condição para que ela não seja sempre avaliada como "false"; algum código subsequente nunca é executado.	15min	Maior	Aberto
13	"tryThen" declarado na linha 1547 espera 4 argumentos, mas 5 foram fornecidos. Remova essa instrução "throw" desse bloco "finally".	10min	Crítico	Aberto
14	Remova essa instrução "throw" desse bloco "finally".	30min	Maior	Aberto

Tabela 8 – Relatório de Vulnerabilidades SonarQube

Vulnerabilidade	Tipo	Tempo de correção	Gravidade	Status
1	Torne este endereço IP "127.0.0.1" configurável.	30min	Menor	Aberto

Tabela 9 – Relatório de *Code Smells* SonarQube

Code Smells	Tipo	Tempo de correção	Gravidade	Status
1	Remova este código comentado.	5min	Maior	Aberto
2	Remova este código comentado.	5min	Maior	Aberto
3	Remova este código comentado.	5min	Maior	Aberto
4	Remova o uso de todos os operadores de vírgula nesta expressão.	5min	Maior	Aberto
5	Renomear "hooks" como esse nome já é usado na declaração na linha 3099.	20min	Maior	Aberto
6	Use o conteúdo desta coleção ou remova a coleção.	2min	Maior	Aberto
7	Remova este código comentado.	5min	Maior	Aberto
8	Remova a variável local não usada "other_list".	5min	Menor	Aberto
9	Remova a variável local não usada "other_list".	5min	Menor	Aberto
10	Remova a variável local não usada "other_list".	5min	Menor	Aberto
11	Remova a variável local não usada "item1".	5min	Menor	Aberto
12	Remova este código comentado.	5min	Maior	Aberto
13	Remova este código comentado.	5min	Maior	Aberto
14	Remova este código comentado.	5min	Maior	Aberto

Tabela 10 – Relatório de Duplicidades SonarQube

Duplicidades	Arquivo	Linhas Duplicadas (%)	Linhas Duplicadas (%)	Arquivo duplicados	Blocos duplicados	Avaliação
1	lists/static/bootstrap/js/bootstrap.js	99,4%	3921	1	2	A
2	lists/static/bootstrap/js/bootstrap.bundle.js	61,0%	3944	1	3	A
3	lists/static/tests/qunit-2.8.0.js	0,4%	23	1	1	A
4	lists/migrations/0001_initial.py	0	0	0	0	A
5	lists/migrations/0002_item_text.py	0	0	0	0	A
6	lists/migrations/0003_auto_20181022_2255.py	0	0	0	0	A
7	functional_tests/__init__.py	0	0	0	0	A
8	lists/__init__.py	0	0	0	0	A
9	lists/tests/__init__.py	0	0	0	0	A
10	lists/migrations/__init__.py	0	0	0	0	A
11	superlists/__init__.py	0	0	0	0	A
12	lists/admin.py	0	0	0	0	A
13	lists/apps.py	0	0	0	0	A
14	functional_tests/base.py	0	0	0	0	A
15	lists/static/bootstrap/js/bootstrap.bundle.min.js	0	0	0	0	A
16	lists/static/bootstrap/js/bootstrap.min.js	0	0	0	0	A
17	lists/forms.py	0	0	0	0	A
18	functional_tests.py	0	0	0	0	A
19	lists/static/jquery-3.3.1.min.js	0	0	0	0	A
20	manage.py	0	0	0	0	A
21	lists/models.py	0	0	0	0	A
22	superlists/settings.py	0	0	0	0	A
23	lists/tests/test_forms.py	0	0	0	0	A
24	functional_tests/test_layout_and_styling.py	0	0	0	0	A
25	functional_tests/test_list_item_validation.py	0	0	0	0	A
26	lists/tests/test_models.py	0	0	0	0	A
27	functional_tests/test_simple_list_creation.py	0	0	0	0	A
28	lists/tests/test_views.py	0	0	0	0	A
29	lists/urls.py	0	0	0	0	A
30	superlists/urls.py	0	0	0	0	A
31	lists/views.py	0	0	0	0	A
32	superlists/wsgi.py	0	0	0	0	A

Tabela 11 – Relatório de Cobertura SonarQube

Cobertura	Arquivo	Cobertura	Linhas descobertas	Condições descobertas
1	lists/migrations/0001_initial.py	0,0%	5	—
2	lists/migrations/0002_item_text.py	0,0%	4	—
3	lists/migrations/0003_auto_20181022_2255.py	0,0%	5	—
4	lists/admin.py	0,0%	1	—
5	lists/apps.py	0,0%	3	—
6	functional_tests/base.py	0,0%	34	—
7	lists/static/bootstrap/js/bootstrap.bundle.js	0,0%	2443	—
8	lists/static/bootstrap/js/bootstrap.js	0,0%	1743	—
9	lists/forms.py	0,0%	26	—
10	functional_tests.py	0,0%	33	—
11	manage.py	0,0%	13	—
12	lists/models.py	0,0%	13	—
13	lists/static/tests/qunit-2.8.0.js	0,0%	2449	—
14	superlists/settings.py	0,0%	19	—
15	lists/tests/test_forms.py	0,0%	40	—
16	functional_tests/test_list_item_validation.py	0,0%	26	—
17	lists/tests/test_models.py	0,0%	44	—
18	functional_tests/test_simple_list_creation.py	0,0%	44	—
19	lists/tests/test_views.py	0,0%	99	—
20	lists/urls.py	0,0%	3	—
21	superlists/urls.py	0,0%	5	—
22	lists/views.py	0,0%	21	—
23	superlists/wsgi.py	0,0%	4	—

4.1.5 Integração com Docker para *build* da imagem da aplicação

Conforme detalhado anteriormente no Algoritmo 4 apresentado na seção 3.3.3, os passos necessários para construção da imagem, dar-se-a por meio do arquivo *Dockerfile*, que recebe as instruções de como a imagem será construída. O arquivo *Dockerfile* encontra-se no mesmo repositório da aplicação, onde poderá ser acessado pelo servidor de integração Jenkins. O segundo arquivo necessário para executar o contêiner é o *docker-compose* também armazenado no repositório. Esse arquivo permite que a imagem da aplicação seja inicializada.

O tempo de execução de todo o cenário depende em que estágio da configuração está, nesse trabalho constatou-se que o primeiro *build* é o mais demorado, visto que faz todas as verificações necessárias. Tal como baixar o repositório, executar os testes, comunicar com o servidor de análise do código, gerar imagem da aplicação e por fim fazer o *build* da mesma. Notou-se que a primeira execução leva em média 15min a 20 min. Já os demais *builds*, esse valor vai diminuindo progressivamente, já que a maioria das dependências estão no *workspace* do Jenkins. O menor tempo verificado foi de 6 min 44 sec.

Ao completar todas essas etapas do pipeline é possível notar que a volta nas atividades de *DevOps* são realizadas de forma constante, conforme apresentado na Figura 4 do capítulo 2. A execução de cada atividade do ciclo *DevOps* aplicada ao cenário estudado é descrito a seguir:

- **Planejar** - Realizada previamente na definição do problema a ser resolvido;
- **Codificar** - Executada a partir da codificação em pequenas unidades da página web, com o uso do *TDD* e seu ciclo contínuo de escrever teste/rodar o teste/refatorar o código;
- **Construir** - Foi realizada a partir da integração do *Dockerfile* e *docker-compose*, onde contruiu-se a imagem da aplicação;
- **Testar** - Feita constantemente através do servidor de integração Jenkins, onde a cada *commit* identificado pelo Jenkins os testes disparavam automaticamente;
- **Lançar e Operar** - Atividades realizadas a partir da implantação da aplicação de forma automatizada, usando a imagem docker construída foi possível hospedar a imagem em servidor de testes;
- **Monitorar** - Alcançada através do servidor de inspeção de código - SonarQube - onde é verificado as métricas de análise do código e também através dos *logs* emitidos pelo Jenkins.

5 Considerações Finais

A finalidade desse trabalho consistiu na aplicação da técnica *TDD* para hospedagem de sistemas *web*, utilizando a filosofia *DevOps*. Para o correto desenvolvimento de *software*, além de conhecimento técnico, organização e atenção, é importante antecipar-se as falhas, fazendo isso de forma fácil e ágil, executando testes a todo momento para que isso fosse viável, a utilização de testes dos testes automatizados tornam-se imprescindíveis.

Durante a automação dos testes foi investido um tempo considerável escrevendo testes unitários que examinava cada parte do código, bem como os testes funcionais onde se verificava as possíveis entradas não permitidas, isso permitiu que as alterações no código trouxesse mais segurança durante cada refatoração. O tempo gasto durante os testes permitiu visualizar a importância dos testes, pois é possível diminuir o tempo com correções e identificação dos erros.

E com a automatização da execução dos testes através do Jenkins foi possível manter uma verificação constante das mudanças no repositório, sempre que constatava alguma alteração no código os testes eram acionados automaticamente, facilitando assim a tarefa manual de executar comandos frequentemente para saber se a aplicação está com algum problema.

Para aplicar o *TDD* é fundamental ter disciplina para não abandonar a técnica no meio do desenvolvimento por achar algo repetitivo e difícil, esse é um dos pontos que mais fazem diferença para alcançar qualidade no código e no desenvolvimento dos testes contínuos.

Desta maneira, reduzir as taxas de defeitos do código podem ser diminuídas através da inspeção contínua da qualidade do código por meio de soluções, tal como o SonarQube, que fornece análise estatística do código, demonstrando vulnerabilidades, más práticas de codificação e até possíveis *bugs*. A utilização dessa ferramenta pode impedir que novas versões da aplicação que não atendam aos critérios mínimos de qualidade vá para o ambiente de produção.

Depois de entender os conceitos visto no capítulo 2 e de executar os mesmo no capítulo 3 observou-se que esse modelo de prover hospedagem de sistemas *web* com qualidade e continuamente, fornece uma solução ágil e prática para as equipes que desejam iniciar o processo de migração da plataforma tradicional de desenvolvimento para o modelo *DevOps*. Pois, trabalhar em ambiente padronizado onde todas as ferramentas são comuns entre a equipe, permite maior eficiência e agilidade na entrega de produtos e serviços com qualidade.

Na seção seguinte são apresentadas as contribuições e limitações deste trabalho, assim como os possíveis trabalhos futuros.

5.1 Contribuições e Limitações

As principais contribuições deste trabalho foram:

- A aprendizagem de conceitos novos, tal como *TDD* e *DevOps*; E ferramentas tal como Jenkins, SonarQube e Docker, foram muito importantes para a realização desse projeto. E permitiu conhecer uma das áreas de atuação da TI, que é pouco explorada no ambiente acadêmico;
- A implementação de uma aplicação *web* com a linguagem de programação Python, utilizando *framework* Django, para exemplificação da técnica de *TDD*;
- A modelagem do cenário de integração contínua para testes e implantação de sistemas *web*;
- A exemplificação do uso da plataforma de inspeção de qualidade de código: SonarQube, para dar suporte à implementação do cenário;
- A exemplificação do uso do Docker para apoio à implantação da aplicação, utilizando *Dockerfile* e *docker-compose*;
- A simulação de um ambiente de integração contínua, para exemplificar a integração de cada ferramentas e técnicas estudadas durante a pesquisa;

As principais limitações e dificuldades deste trabalho foram:

- Para aplicar os testes no cenário, foi preciso construir todos os serviços necessário para integração, o que levou bastante tempo estudando e instalando as ferramentas para integrá-las e só após isso iniciar os testes;
- A implementação do cenário proposto foi feito utilizando servidor dos laboratórios da UNITINS o que limitava a pesquisa, pois nem sempre os serviços estavam ativos, devido a constantes quedas de energia;
- O uso por tempo limitado dos laboratórios foi uma dificuldade também, pois às vezes o trabalho precisava ser interrompido ou ficava em execução, não sendo possível saber o resultado daquele execução de forma imediata;
- Foi disponibilizado acesso VPN a rede da UNITINS, mas o acesso era bem lento e dificultava bastante elaborar integração entre serviços que precisavam baixar pacotes.

5.2 Trabalhos futuros

Pretende-se como trabalhos futuros:

- Modificar a estrutura do cenário utilizado no trabalho, para *cloud computing* privada. Definir novos parâmetros para os testes, utilizando como base para os testes, um ambiente de produção;
- Utilizar ferramentas de IA existentes para definir qual é o número mínimo de testes para descobrir se as alterações em parte do código é algo bom ou mal;
- Integrar diferentes tecnologias de infraestrutura como código para provimento do ambiente proposto (Vagrant, Chef, Puppet, Ansible, Kubernetes);
- Realizar um estudo comparativo entre ferramentas de inspeção contínua da qualidade do código;
- Aplicar a técnica de *Behaviour Driven Development (BDD)* de forma a complementar e ampliar o uso de TDD.

Referências

AIELLO, B.; SACHS, L. Devops best practices, part 6: Use devops to drive quality assurance and testing. *developerWorks*, n. 6, 2014. Acesso em: 15 mai. 2018.

ANICHE, M.; CARDOSO, A. *Test-Driven Development: Teste e Design no Mundo Real com PHP*. [S.l.]: Casa do Código, 2015. ISBN 978-8555190612.

AWS. *O que é DevOps?* 2018. Disponível em: <<https://aws.amazon.com/pt/devops/what-is-devops/>>. Acesso em: 01 mai. 2018.

BECK, K. *Test Driven Development: By Example*. [S.l.]: Addilson-Wesley, 2012.

BERNADO, P.; KON, F. A importância dos Testes Automatizados. *Revista Engenharia de Software Magazine*, p. 54–57, 2008.

BOAGLIO, F. *Jenkins: automatize tudo sem complicações*. [S.l.]: Casa do Código, 2016.

DEBOIS, P. *What Is This Devops Thing, Anyway?* 2010. Disponível em: <<http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/>>. Acesso em: 07 abr. 2018.

DEBOIS, P. *Devops Areas - Codifying devops practices*. 2012. Disponível em: <<http://www.jedi.be/blog/2012/05/12/codifying-devops-area-practices/>>. Acesso em: 07 abr. 2018.

DIAS, G. (Ed.). *Curso: DevOps Essentials. Aula 01 – A Cultura DevOps*. São Paulo: [s.n.], 2018.

DJANGO, P. *Documentação do Django*. 2018. Disponível em: <<https://docs.djangoproject.com/pt-br/2.1/>>. Acesso em: 25 Nov. 2018.

DUNNE, K. *Test-First Development: Processes and Tools for Success*. 2016. Disponível em: <<https://devops.com/test-first-development-processes-tools-success/>>. Acesso em: 12 mai. 2018.

ELIAV, R. *Dissecting the Role of QA Engineers and Developers in Functional Testing*. 2018. Disponível em: <<https://devops.com/dissecting-the-role-of-qa-engineers-and-developers-in-functional-testing/>>. Acesso em: 31 mai. 2018.

FILHO, W. d. P. P. *Engenharia de Software - Fundamentos, Métodos e Padrões*. [S.l.]: LTC, 2009. ISBN 978-85-216-1650-4.

GASKELL, G. et al. *Jenkins User Documentation*. 2018. Disponível em: <<https://jenkins.io/doc/>>. Acesso em: 15 out. 2018.

GAŁĘZOWSKI, G. *Test-Driven Development: Extensive Tutorial*. 2018. Disponível em: <<https://github.com/grzesiek-galezowski/tdd-ebook>>. Acesso em: 23 mar. 2018.

GEER, D. *QA in the DevOps Era*. 2016. Disponível em: <<https://devops.com/qa-devops-era/>>. Acesso em: 27 mai. 2018.

- GIL, A. C. *Métodos e técnicas de pesquisa social*. [S.l.]: Atlas, 2008. ISBN 978-85-224-5142-5.
- HÜTTERMANN, M. *DevOps for Developers*. [S.l.]: Apress, 2012.
- KERSTEN, N. *What is DevOps?* 2017. Disponível em: <<https://puppet.com/blog/what-is-devops>>. Acesso em: 15 mai. 2018.
- LEE, J. *A Practical Guide to Getting Started with DevOps*. [S.l.]: Level Up, 2017.
- LESZKO, R. *Continuous Delivery with Docker and Jenkins*. [S.l.]: Packt Publishing, 2017. ISBN B0756FCZQY.
- MACVITTIE, D. *Test Driven Development, Continuous Test and Code Quality*. 2017. Disponível em: <<https://devops.com/test-driven-development-continuous-test-code-quality/>>. Acesso em: 27 mai. 2018.
- MEZAK, S. *QA/Testing and DevOps: Partners, Not Adversaries*. 2017. Disponível em: <<https://devops.com/qatesting-devops-partners-not-adversaries/>>. Acesso em: 24 mai. 2018.
- MOUAT, A. *Usando Docker - Desenvolvendo e Implantando Software com Contêineres*. [S.l.]: Novatec, 2016. ISBN 978-85-7522-492-2.
- MYERS, G. J. *The art of software testing*. [S.l.]: John Wiley & Sons Inc, 2004. ISBN 978-1-118-13313-2.
- PERCIVAL, H. J. *TDD com Python - Siga o Bode dos Testes: Usando Django, Selenium e Javascript*. [S.l.]: Novatec, 2017. ISBN 978-85-7522-642-1.
- PFLIEGER, S. L. *Engenharia de Software - Teoria e Prática*. [S.l.]: Pearson Prentice Hall, 2004. ISBN 978-85-87918-31-4.
- PICOLO, M. *Test Driven Development*. 2017. Disponível em: <<https://github.com/fga-gpp-mds/A-Disciplina/wiki/Test-Driven-Development>>. Acesso em: 30 mar. 2018.
- PRESSMAN, R. S. *Engenharia de Software*. [S.l.]: AMGH Editora LTDA, 2011. ISBN 978-0-073-37597-7.
- PRODANOV, C. C.; FREITAS, E. C. *Metodologia do trabalho científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico*. [S.l.]: Universidade Feevale, 2013. ISBN 978-85-7717-158-3.
- PUPPET. *2017 State of DevOps Report*. 2017. Disponível em: <<https://puppet.com/resources/whitepaper/state-of-devops-report>>. Acesso em: 01 mai. 2018.
- RILEY, C. *You Asked for Quality, You Got Alienated QA*. 2014. Disponível em: <<https://devops.com/asked-quality-got-alienated-qa/>>. Acesso em: 27 mai. 2018.
- S.A, S. S. *SonarQube Docs 7.4 - Metric Definitions*. 2018. Disponível em: <<https://docs.sonarqube.org/latest/user-guide/metric-definitions/>>. Acesso em: 24 nov. 2018.
- SADASIVAN, K. *DevOps*. [S.l.]: Happiest Minds Technologies Pvt. Ltd, 2014.
- SATO, D. *DevOps na prática: entrega de software confiável e automatizada*. [S.l.]: Casa do Código, 2013.

- SHARMA, S.; COYNE, B. *DevOps for Dummies - A Wiley Brand. 3rd IBM Limited Edition*. [S.l.]: John Wiley & Sons Inc, 2017. ISBN 978-1-119-41588-6.
- SHIMEL, A. *Continuous Testing The Final Frontier of Continuous Delivery*. [S.l.]: CA Technologies, 2018.
- SLYNGSTAD, O. P. N. et al. The impact of test driven development on the evolution of a reusable framework of components 150; an industrial case study. In: *2008 The Third International Conference on Software Engineering Advances*. [S.l.: s.n.], 2008. p. 214–223.
- SOMMERVILLE, I. *Engenharia de Software*. [S.l.]: Pearson Prentice Hall, 2011. ISBN 978-85-7936-108-1.
- TRODO, L. D. *Uso de Métricas nos Testes de Software*. [S.l.]: Universidade Federal do Rio Grande do Sul, 2009.
- WILLIS, J.; EDWARDS, D. *DevOps Culture (Part 1)*. 2010. Disponível em: <<http://itrevolution.com/devops-culture-part-1>>. Acesso em: 07 abr. 2018.
- YIN, R. K. *Estudo de caso: planejamento e métodos*. [S.l.]: Bookman, 2001. ISBN 85-7307-852-9.

Apêndices

APÊNDICE A – INSTALAÇÃO E CONFIGURAÇÃO DO AMBIENTE

O processo de instalação das ferramentas é detalhado nas seções subsequentes.

A.1 Instalando o Docker

Primeiramente atualize os repositório do sistema operacional com o comando abaixo:

```
# sudo apt-get update
```

Instale a versão mais recente do Docker CE, com o comando:

```
# sudo apt-get install docker-ce
```

A.2 Instalando o Jenkins via Docker

Após instalação do Docker, puxe a imagem oficial do jenkins do repositório do Docker, com o comando a seguir:

```
# docker pull jenkins/jenkins:lts
```

Em seguida, execute um contêiner usando essa imagem e mapeie o diretório de dados do contêiner para o *host*; no exemplo abaixo o container é mapeado em `/var/jenkins_home` onde será armazenado o espaço de trabalho do Jenkins, incluindo *plugins* e configuração. A porta 8080 do Jenkins também estará acessível ao *host* pela 50000.

```
# docker run -p 8080:8080 -p 50000:50000 jenkins/jenkins:lts
```

Para poder fazer o primeiro acesso a nova instalação do servidor Jenkins, obtenha a senha inicial com o comando:

```
# sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

Digite no navegador o ip ou dns do servidor Jenkins e a porta da seguinte forma `http://ip:8080`

Cole a senha inicial no campo Administrator password e clique em Continue.

Clique em *Install suggested plugins*.

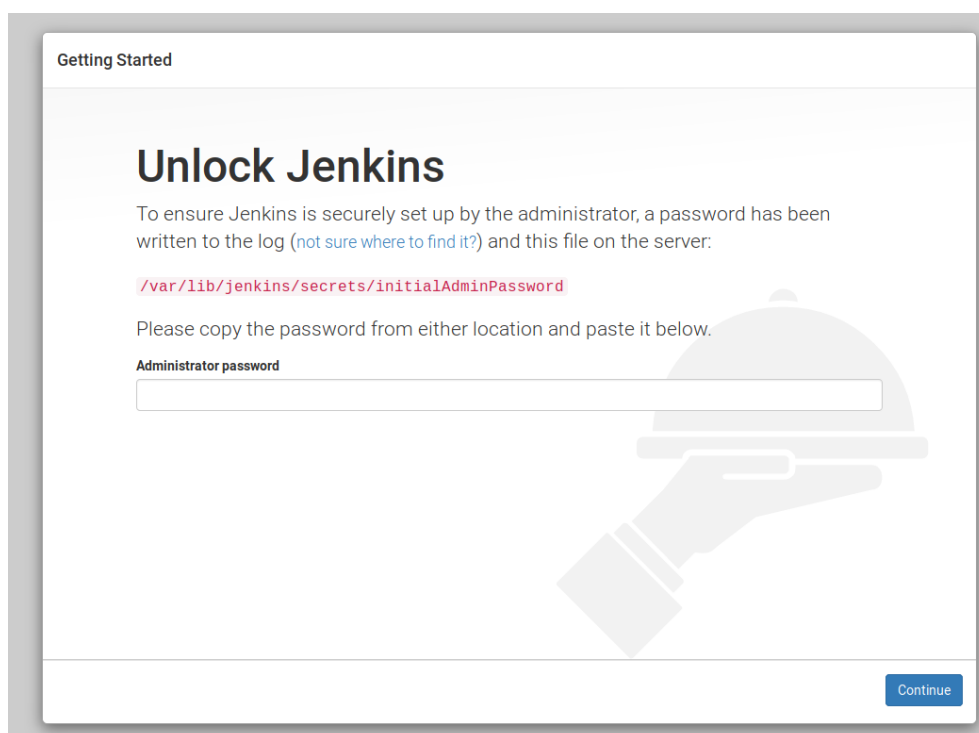


Figura 15 – Confirmando acesso como administrador
Fonte: Autoria própria.

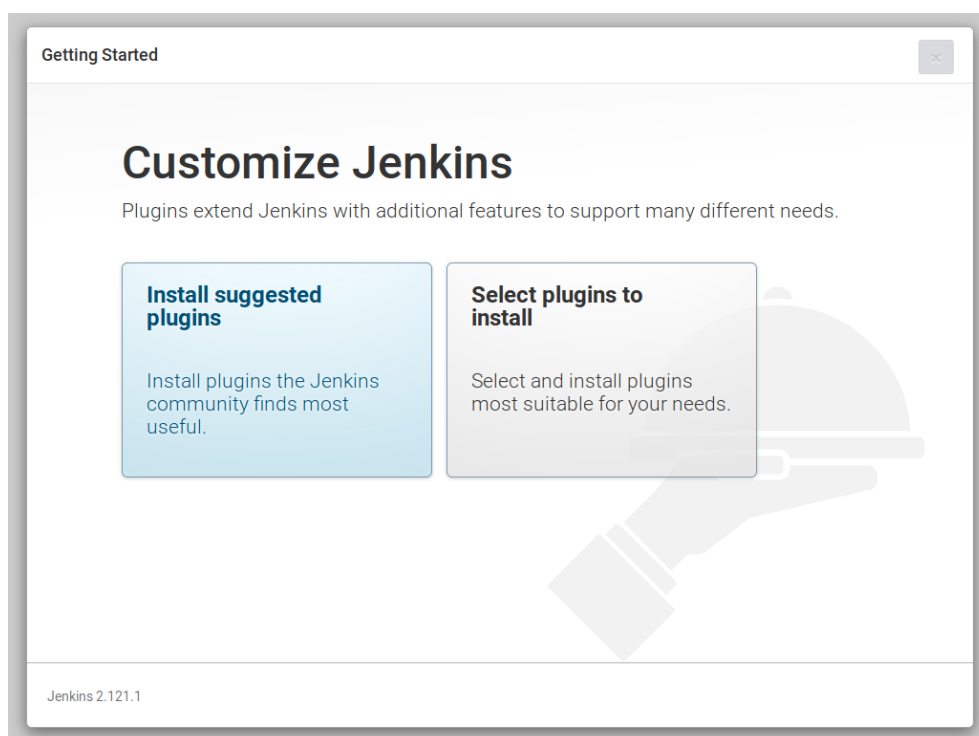


Figura 16 – Tela de seleção de *plugins*
Fonte: Autoria própria.

Aguarde a instalação dos *plugins*.

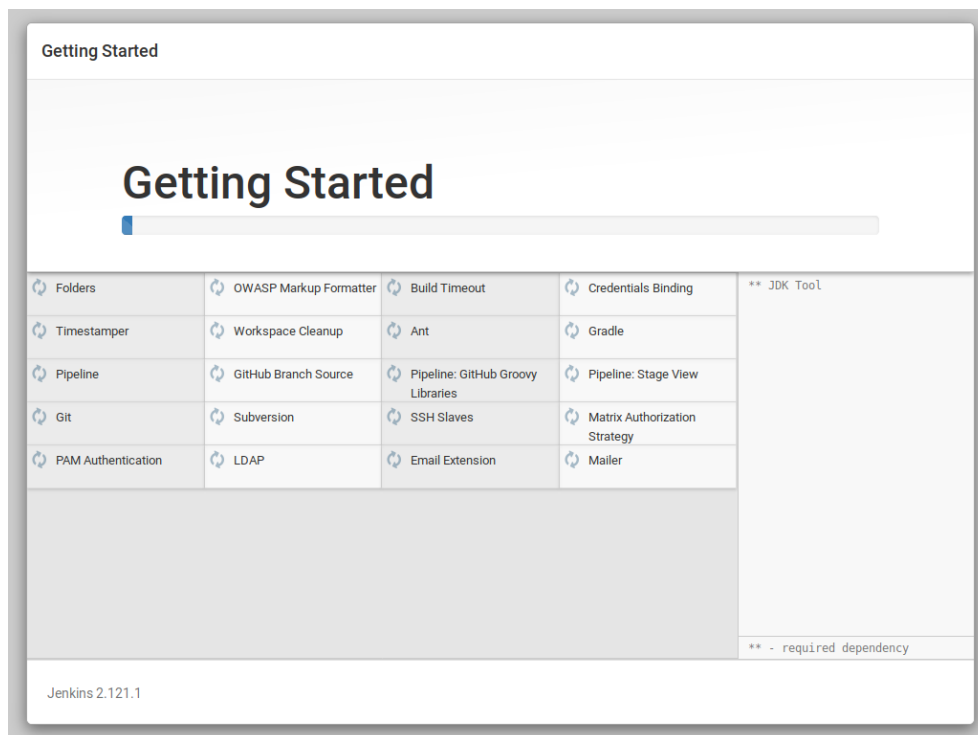


Figura 17 – Progresso da instalação de *plugins* sugeridos

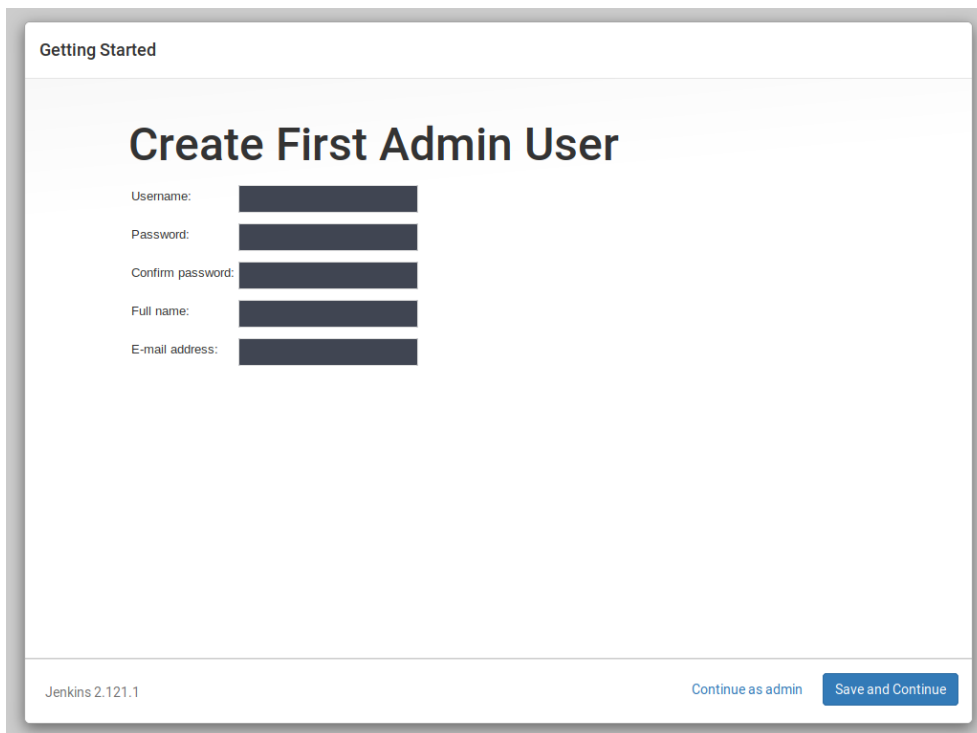
Fonte: Autoria própria.

Assim que encerradas as instalações é possível criar um usuário administrador. Preencha todos os campos e clique em *Save and Continue* ou *continue* como admin e clique em *Continue as admin*.

Altere o *host* e a porta se desejar e clique em *Save and Finish*.

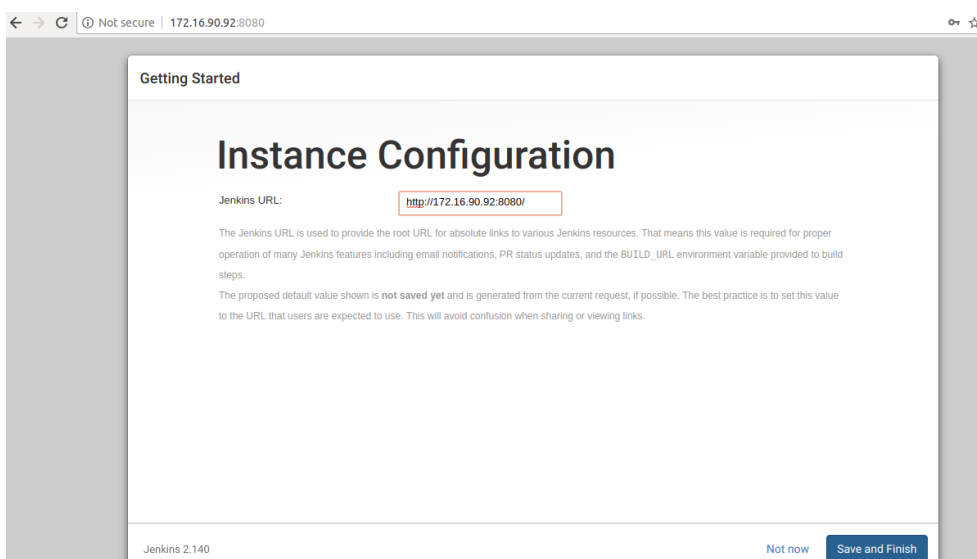
Caso tenha pulado alguma das etapas de configuração mensagens em amarelo aparecerão nesta tela. Clique em *Start using Jenkins*.

Caso tenha criado o usuário administrador ele será exibido no canto superior direito, caso contrário aparecerá admin. A senha do usuário admin é a senha inicial exibida em etapa anterior neste tutorial.



The screenshot shows the 'Getting Started' page of Jenkins 2.121.1. The main heading is 'Create First Admin User'. Below it are five input fields: 'Username:', 'Password:', 'Confirm password:', 'Full name:', and 'E-mail address:'. At the bottom right, there are two buttons: 'Continue as admin' and 'Save and Continue'.

Figura 18 – Tela criação de usuário administrador
Fonte: Autoria própria.



The screenshot shows the 'Getting Started' page of Jenkins 2.140. The main heading is 'Instance Configuration'. Below it is a 'Jenkins URL:' label followed by a text input field containing 'http://172.16.90.92:8080/'. Below the input field is a paragraph of text explaining the Jenkins URL. At the bottom right, there are two buttons: 'Not now' and 'Save and Finish'.

Figura 19 – Tela de definição da URL do Jenkins
Fonte: Autoria própria.

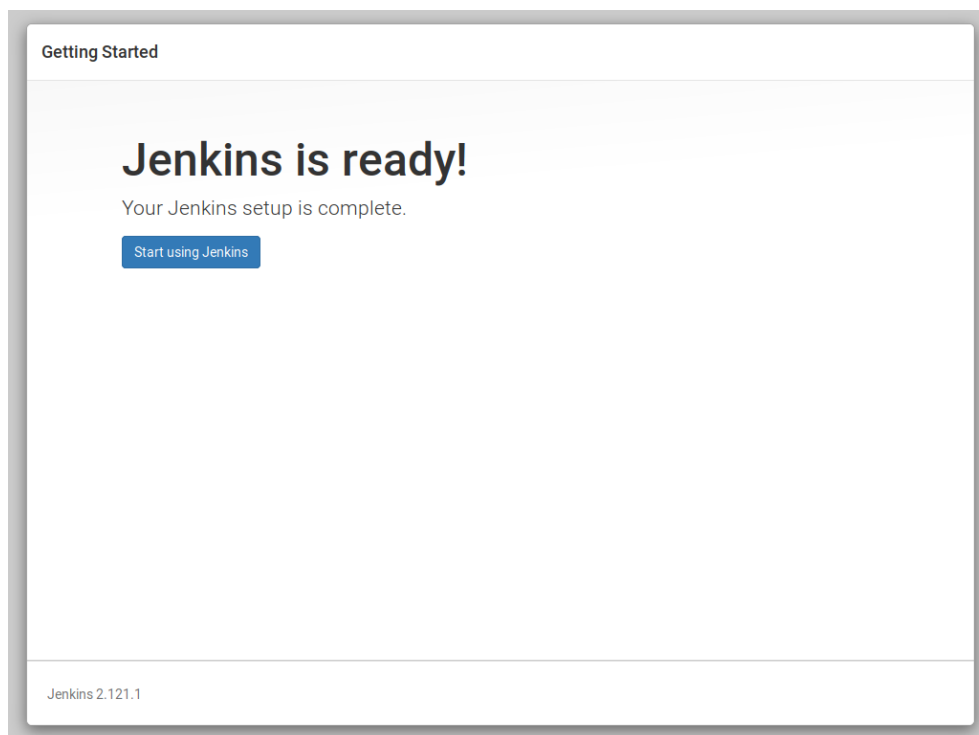


Figura 20 – Tela de finalização das configurações
Fonte: Autoria própria.

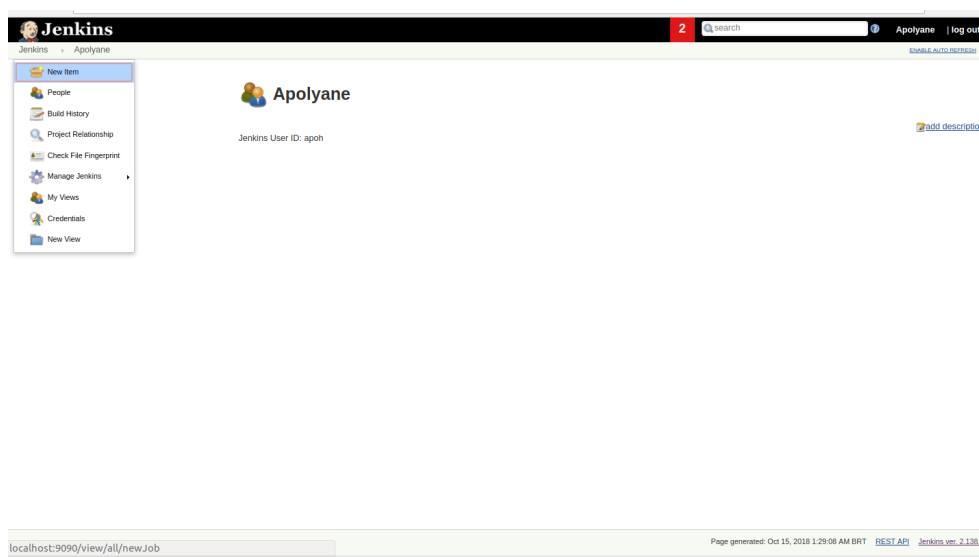


Figura 21 – Tela inicial do Jenkins
Fonte: Autoria própria.

A.3 Configurando *Job* integrando Jenkins com Github

1) Primeiramente será necessário instalar e configurar o plugin do Github. Através da página inicial do Jenkins clique em *Manage Jenkins* -> *Manage Plugins*:

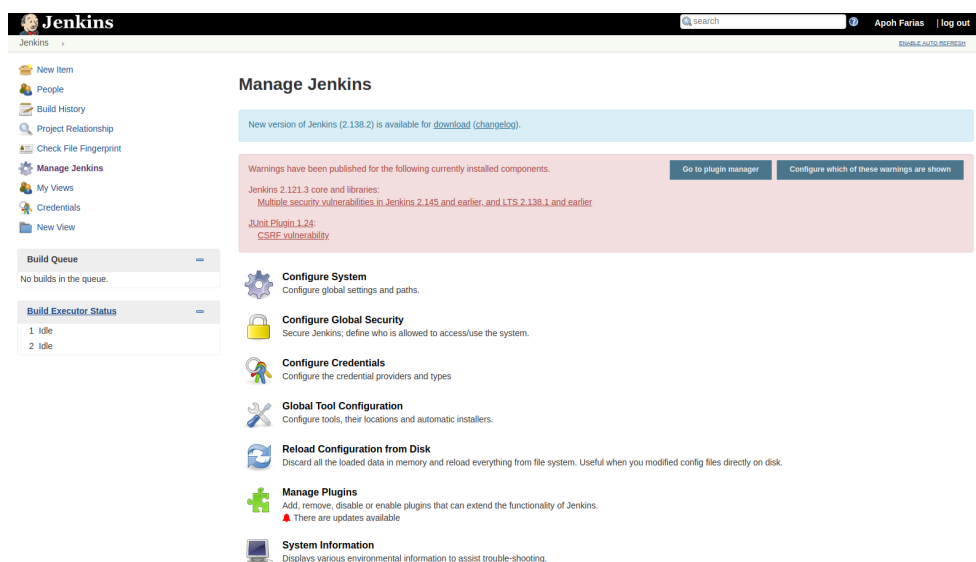


Figura 22 – Página de Gerenciamento do Jenkins
Fonte: Autoria própria.

Na aba *Available*, habilite o Git Plugin e GitHub, depois clique em *Download now and install after restart*:

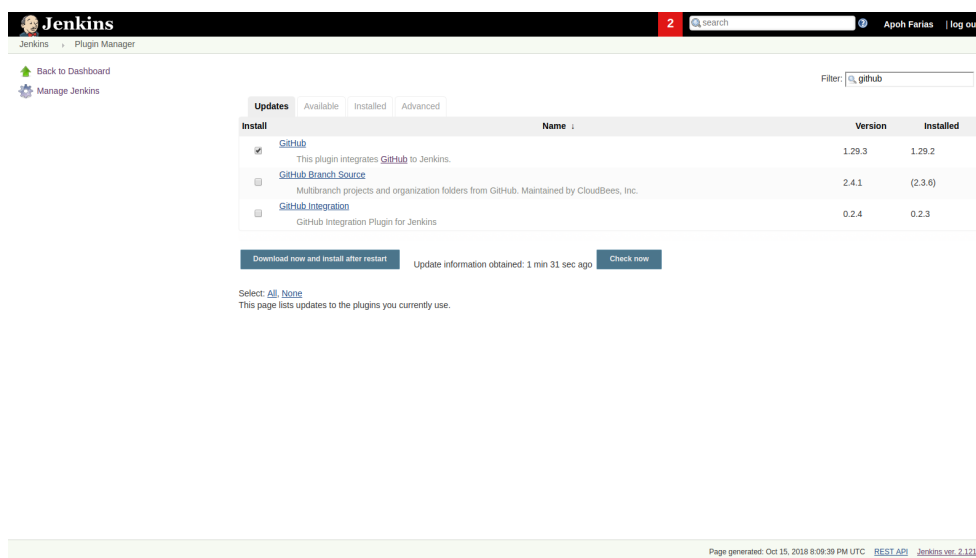
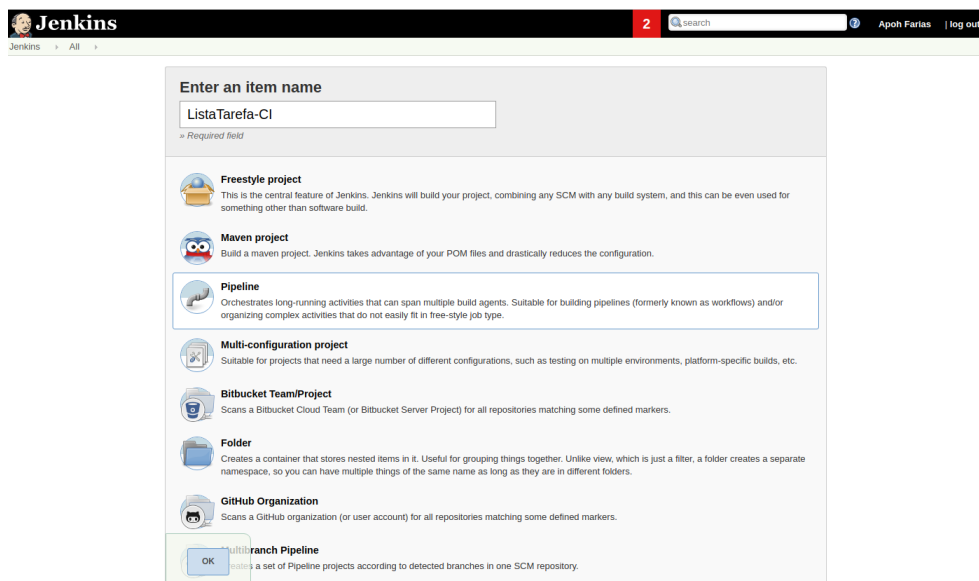


Figura 23 – Página de Gerenciamento de *Plugins*
Fonte: Autoria própria.

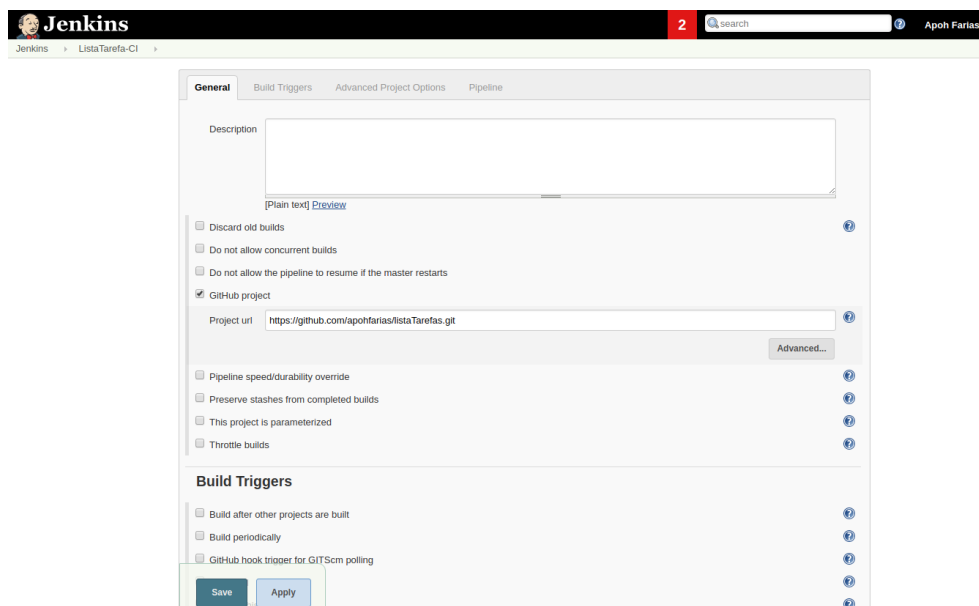
2) Criando um *Job* no Jenkins

Vá para a tela inicial do Jenkins e clique em *"New Item"*. Na próxima tela, crie um nome para o *Job*, escolha a opção *'Freestyle project'*. Depois clique em *"OK"*.

Figura 24 – Página de Criação de novo *Job*

Fonte: Autoria própria.

Na sessão General clique em *GitHub Project*. Em *Project URL* coloque a url do seu projeto no GitHub. Clique em *Save*.

Figura 25 – Página de Configuração do *Job*

Fonte: Autoria própria.

Na tela seguinte, execute o *job* e solicite a criação de um *build*, clicando no ícone "*Build Now*" do lado esquerdo:

Após clicar nessa opção, à esquerda aparecerá no *Build History* um número do *build* com # , que é um valor crescente e sequencial. Ao clicar nesse número, as informações do *Build* são apresentadas, como o tempo de execução do *build*:

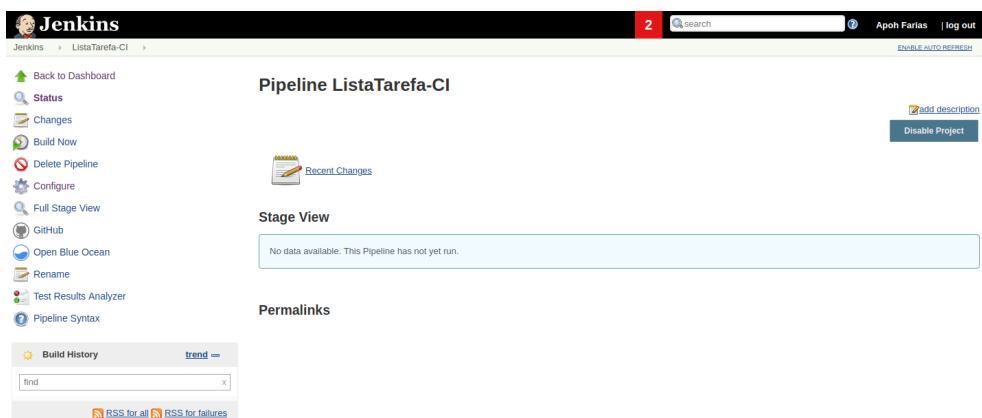


Figura 26 – Página do *workspace* do *Job* recém criado
Fonte: Autoria própria.

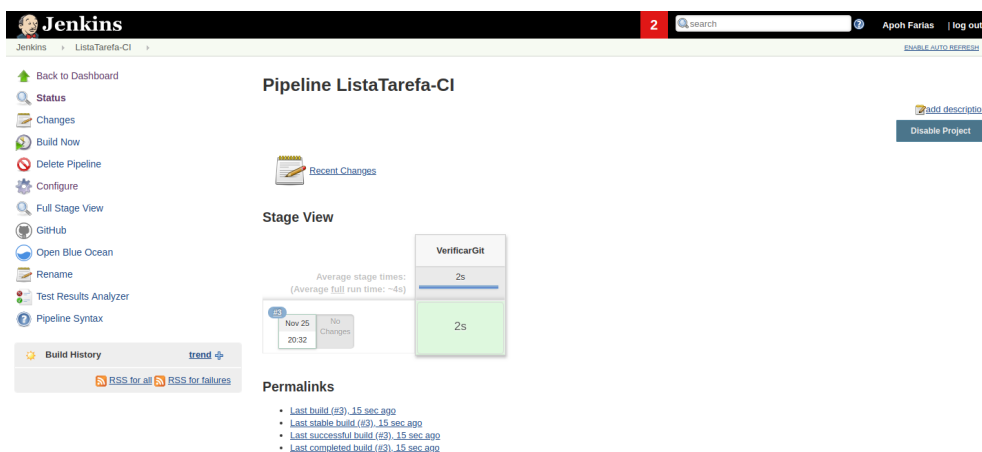


Figura 27 – Página de detalhes do *job*
Fonte: Autoria própria.

Para ver as informações do console clique em "*Console Output*", será exibido o texto da saída do *build* e atualizado automaticamente:

Se tudo ocorreu bem, o final da saída será exibido algo assim:

```
[Pipeline] // stage
[Pipeline]
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

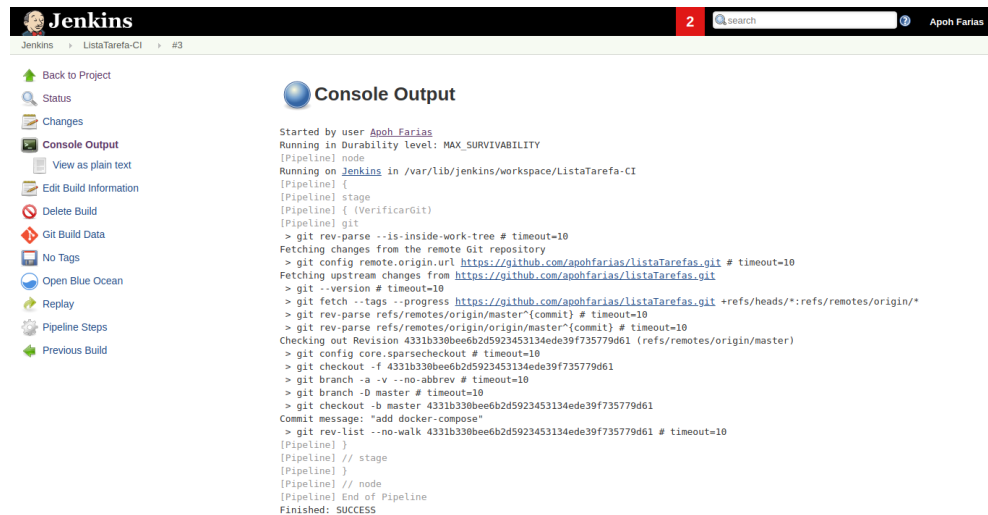


Figura 28 – Página da saída do console
Fonte: Autoria própria.

A.4 Instalando o SonarQube via Docker

Crie uma pasta chamada sonar em /data

```
# mkdir -p /data/sonar
```

Crie o script do compose dentro da pasta recém criada

```
# nano docker-compose.yml
```

Execute o script do compose:

```
# sudo docker-compose up -d
```

Vá ao navegador e acesse a aplicação <http://ip:9000>

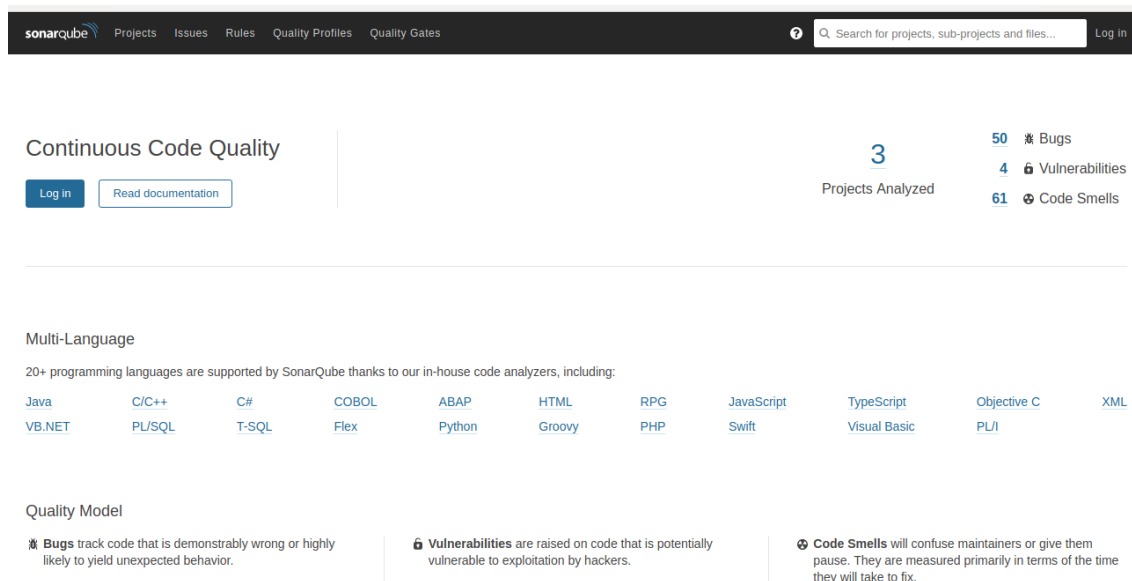


Figura 29 – Página Inicial do SonarQube
Fonte: Autoria própria.

A.4.1 Configuração SonarQube Scanner no pipeline

Os parâmetros para configurar a análise do projeto deve ser definidos em três locais, primeiramente configurando o servidor. Vá em *Manage Jenkins > Configure System > SonarQube servers* e coloque o nome para o servidor, o endereço URL e o *token*, conforme Figura 30:

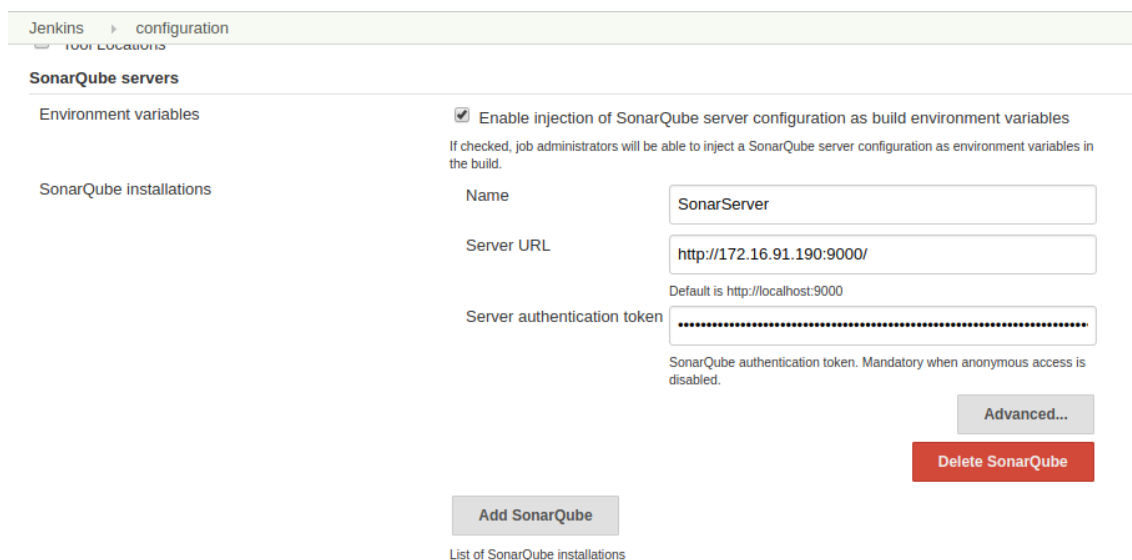


Figura 30 – Configurar SonarQube Servers
Fonte: Autoria própria.

A segunda etapa é definir o SonarQube Scanner, que irá escanear o projeto e fazer análise do projeto. Vá em *Manage Jenkins > Global Tool Configuration > SonarQube Scanner* e adicione um nome para o escaner, marque a opção *Install automatically* e escolha a versão que será usada, conforme Figura 31:

The screenshot shows the Jenkins 'Global Tool Configuration' page for 'SonarQube Scanner'. At the top, there's a breadcrumb 'Jenkins > Global Tool Configuration'. Below the title 'SonarQube Scanner', there's a section 'SonarQube Scanner installations'. It features an 'Add SonarQube Scanner' button. Below this, a table lists existing installations. One installation is shown with the name 'SonarQube Scanner'. To the right of the table is a 'Delete Installer' button. Below the table, there's a checkbox for 'Install automatically' which is checked. Below that, there's a section 'Install from Maven Central' with a 'Version' dropdown menu set to 'SonarQube Scanner 2.3'. To the right of this section is a 'Delete SonarQube Scanner' button. At the bottom, there's an 'Add Installer' button and another 'Add SonarQube Scanner' button. A small text at the bottom reads 'List of SonarQube Scanner installations on this system'.

Figura 31 – Configurar SonarQube *Scanner*

Fonte: Autoria própria.

E por fim adicionar ao projeto os parâmetros de análise, conforme Figura 32.

- *sonar.projectKey*: A chave exclusiva do projeto. Caracteres permitidos são: letras, números, -, _, . e :, com pelo menos um não-dígito.
- *sonar.sources*: Caminho para o diretório que possui os arquivos de origem.
- *sonar.host.url*: URL do servidor
- *sonar.login*: *Login* ou *token* de autenticação de um usuário do SonarQube com a permissão de executar análise no projeto.

The screenshot shows the Jenkins configuration interface for a build step named 'Execute SonarQube Scanner'. The page is divided into several tabs: 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'. The 'Build' tab is currently selected. The configuration fields are as follows:

- Task to run:** A text input field.
- JDK:** A dropdown menu set to '(Inherit From Job)'. Below it, a label reads 'JDK to be used for this SonarQube analysis'.
- Path to project properties:** A text input field.
- Analysis properties:** A text area containing the following properties:

```
sonar.projectKey=listatarefa
sonar.sources=/var/lib/jenkins/workspace/listaTarefasCI/
sonar.host.url=http://172.16.91.190:9000
sonar.login=4da5650dddb05e04cab33180e8b454b11ffa0976
```
- Additional arguments:** A text input field containing '-e -X'.
- JVM Options:** A text input field.

At the bottom of the configuration section, there is an 'Add build step' button. Below the configuration section, the 'Post-build Actions' section is visible, containing 'Save' and 'Apply' buttons.

Figura 32 – Configurar projeto com o SonarQube Scanner
Fonte: Autoria própria.

A.5 Script do *docker-compose* SonarQube

Listing A.1 – Script do *docker-compose* SonarQube

```
version: "2"

services:
  sonarqube:
    image: sonarqube
    ports:
      - "9000:9000"
    networks:
      - sonarnet
    environment:
      - SONARQUBE_JDBC_URL=jdbc:postgresql://db:5432/sonar
    volumes:
      - sonarqube_conf:/opt/sonarqube/conf
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_bundled-plugins:/opt/sonarqube/lib/bundled-plugins

  db:
    image: postgres
    networks:
      - sonarnet
    environment:
      - POSTGRES_USER=sonar
      - POSTGRES_PASSWORD=sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data

networks:
  sonarnet:
    driver: bridge

volumes:
  sonarqube_conf:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_bundled-plugins:
  postgresql:
  postgresql_data:
```

A.6 Scripts para containerização da aplicação

Listing A.2 – Script *Dockerfile* para aplicação

```
FROM python:3.6
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
ADD requirements.txt /code/
ADD manage.py /code/
ADD requirements-dev.txt /code/
RUN pip install -r requirements.txt
ADD . /code/
```

Listing A.3 – Script do *docker-compose* da aplicação

```
version: '3'

services:
  db:
    image: postgres
  web:
    build: .
    command: python3.6 manage.py runserver 0.0.0.0:8000
    volumes:
      - ../code
    ports:
      - "8000:8000"
    depends_on:
      - db
```