

UNIVERSIDADE DO EXTREMO SUL CATARINENSE - UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

KEVIN DE FAVERI AGUIAR

**INTEGRAÇÃO DE FERRAMENTAS OPEN-SOURCE PARA ELABORAÇÃO
DE INFRAESTRUTURA PARA IMPLANTAÇÃO CONTÍNUA COM FOCO EM
AMBIENTES DE DESENVOLVIMENTO JAVA**

CRICIÚMA

2018

KEVIN DE FAVERI AGUIAR

**INTEGRAÇÃO DE FERRAMENTAS OPEN-SOURCE PARA ELABORAÇÃO
DE INFRAESTRUTURA PARA IMPLANTAÇÃO CONTÍNUA COM FOCO EM
AMBIENTES DE DESENVOLVIMENTO JAVA**

Trabalho de Conclusão de Curso,
apresentado para obtenção do grau de
Bacharel no curso de Ciência da
Computação da Universidade do Extremo
Sul Catarinense, UNESC.

Orientador: Prof. Esp. Fabrício Giordani

CRICIÚMA

2018

KEVIN DE FAVERI AGUIAR

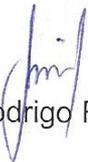
**INTEGRAÇÃO DE FERRAMENTAS OPEN-SOURCE PARA ELABORAÇÃO
DE INFRAESTRUTURA PARA IMPLANTAÇÃO CONTÍNUA COM FOCO EM
AMBIENTES DE DESENVOLVIMENTO JAVA**

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de Bacharel, no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Desenvolvimento para Web.

Criciúma, 28 de Junho de 2018.

BANCA EXAMINADORA


Prof. Fabrício Giordani - Esp - UNESC - Orientador


Prof. Anderson Rodrigo Farias - Esp - SATC


Prof. Gustavo Bisognin - Me. - UNESC

Dedico este trabalho a toda minha família que, com muito carinho e apoio, não mediram esforços para que eu atingisse a graduação.

AGRADECIMENTOS

Quero agradecer primeiramente ao Sicoob Credija, e em especial, a equipe de TI, que me proporcionou um imenso crescimento profissional e intelectual relacionado as áreas de desenvolvimento de software e engenharia de software.

Agradeço a todo o corpo docente do curso de Ciência da Computação pelos conhecimentos repassados e em especial ao meu orientador Fabrício Giordani, que prestou uma tutoria inestimável para a realização deste trabalho.

Por fim gostaria de agradecer aos meus amigos e minha namorada, que como ótimas companhias não me permitiram perder a empolgação com a dificuldades da graduação.

**“A verdadeira sabedoria consiste em saber
que você não sabe nada.”**

Sócrates

RESUMO

O desenvolvimento de software possui diversas etapas que vão desde a sincronização a um repositório de controle de versão até a entrega para o usuário final. O presente trabalho propõe a aplicação da prática ágil de implantação contínua em um ambiente de desenvolvimento Java para web visando eliminar drasticamente processos manuais e automatizar todo o processo de entrega de software. Para definição das etapas de uma infraestrutura de implantação contínua foi realizado um levantamento bibliográfico em conjunto a uma pesquisa de ferramentas *open-source* que atendem aos conceitos que a prática ágil exige. O trabalho trouxe como resultado a automatização de todo o processo de entrega de softwares Java para web em ambiente de produção.

Palavras-chave: Metodologia ágil. Implantação contínua. Automatização. Java para web.

ABSTRACT

Software development has several steps ranging from synchronization to a version control repository to delivery to the final user. The present work proposes the application of the agile practice of continuous implantation in a Java development environment for web to drastically eliminate manual processes and automate the entire process of software delivery. For definitions of the steps of a continuous deployment infrastructure, a bibliographical survey was carried out together with a search for open-source tools that meet the concepts that the agile practice demands. The work resulted in the automation of the entire process of delivering Java software to the web in the production environment.

Keywords: Agile Methodology. Continuous deployment. Automation. Java web.

LISTA DE ILUSTRAÇÕES

Figura 1 – Camadas da engenharia de software.....	117
Figura 2 - Janela do software Virtual Box.....	138
Figura 3 - Código de Instalação do Docker	140
Figura 4 – Comando para execução do contêiner do Jenkins	141
Figura 5 – Comando para adquirir senha inicial de admin para o Jenkins	141
Figura 6 – Instalação do Docker dentro do contêiner do Jenkins.....	141
Figura 7 - Comando de execução do Gitlab.....	142
Figura 8 – Comandos para sincronização do projeto a ser testado com o Gitlab.....	143
Figura 9 – Instalação da ferramenta Maven no Jenkins	144
Figura 10 – Comando para criar uma tela virtualizada no contêiner do Jenkins.....	145
Figura 11 – Comandos para instalação do Google Chrome no contêiner do Jenkins	145
Figura 12 - Dependência do Selenium	145
Figura 13 - Classe de teste funcional	146
Figura 14 – Comando para execução da ferramenta SonarQube.....	147
Figura 15 – Seção de configuração do SonarQube no Jenkins	148
Figura 16 - Comando para execução da ferramenta Nexus.....	149
Figura 17 – Seção de configuração da ferramenta Nexus no Jenkins	149
Figura 18 – Arquivo Dockerfile	150
Figura 19 – Comando para build do contêiner “java-app”	151
Figura 20 – Estrutura inicial do Jenkinsfile	151
Figura 21 – Estágio de build automatizado e implantação em contêiner para testes.....	152
Figura 22 – Estágio para execução de testes unitários e funcionais.....	152
Figura 23 – Estágio para análise da qualidade de código.....	153
Figura 24 – Estágio para salvar executável no repositório de artefatos Nexus	153
Figura 25 – Estágio para implantação em ambiente de produção	154

Figura 26 – Alteração em AcceptanceTests visando um cenário de sucesso na pipeline.....	156
Figura 27 – Pipeline de implantação contínua iniciada pelo Gitlab	156
Figura 28 – Etapa de build automatizado e geração do contêiner para testes	157
Figura 29 – Mensagem de sucesso etapa de build automatizado.....	157
Figura 30 – Etapa de testes unitários e funcionais.....	158
Figura 31 – Impressão no log de alteração efetuada na classe AcceptanceTests.....	158
Figura 32 – Resultado geral dos testes da pipeline.....	158
Figura 33 – Etapa de análise de qualidade de código.....	159
Figura 34 – Análise do projeto Java para web	159
Figura 35 – Etapa para salvar o artefato gerado no Nexus.....	159
Figura 36 – Interface da ferramenta Nexus com o artefato gerado pela pipeline.....	160
Figura 37 – Etapa de implantação em ambiente de produção	160
Figura 38 – Implantação em ambiente de produção com sucesso	161
Figura 39 – Projeto Java para web entregue em ambiente de produção	161
Figura 40 – Alteração em AcceptanceTests visando um cenário de erro na pipeline.....	161
Figura 41 – Erro na etapa de testes unitários e funcionais	162
Figura 42 – Motivo do erro na classe AcceptanceTests.....	162

SUMÁRIO

1 INTRODUÇÃO	113
1.1 OBJETIVO GERAL	114
1.2 OBJETIVOS ESPECÍFICOS	114
1.3 JUSTIFICATIVA	114
1.4 ESTRUTURA DO TRABALHO	115
2 ENGENHARIA DE SOFTWARE	117
2.1 METODOLOGIA ÁGIL DE DESENVOLVIMENTO	118
2.2 VALORES E PRÍNCIPIOS	119
2.3 MÉTODOS ÁGEIS	120
3 INTEGRAÇÃO CONTÍNUA	121
3.1 INFRAESTRUTURA	122
3.2 IMPLANTAÇÃO CONTÍNUA	123
3.3 CONTROLE DE VERSÃO	124
3.4 TESTES	126
3.4.1 Testes Unitários	126
3.4.2 Testes Funcionais	127
3.6 ANÁLISE DE QUALIDADE DE CÓDIGO	127
3.7 <i>BUILD</i> AUTOMATIZADO	128
3.8 REPOSITÓRIO DE ARTEFATOS	129
3.9 SERVIDOR DE INTEGRAÇÃO	130
3.10 IMPLANTAÇÃO EM AMBIENTE DE PRODUÇÃO	131
4 TRABALHOS CORRELATOS	133
4.1 IMPLANTAÇÃO DA INTEGRAÇÃO CONTÍNUA E SEUS BENEFÍCIOS: UM ESTUDO DE CASO	133
4.2 INTEGRAÇÃO CONTÍNUA UTILIZANDO JENKINS	134
4.3 ETAPAS DE METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE ÁGIL BASEADA EM PROGRAMAÇÃO EXTREMA	134
4.4 ELABORAÇÃO DE UM FRAMEWORK PARA INTEGRAÇÃO DE SOFTWARE	135
4.5 UTILIZAÇÃO DO FRAMEWORK EPF COMPOSER PARA A CRIAÇÃO DE UM MODELO DE PROCESSO DE DESENVOLVIMENTO DE SOFTWARE	

BASEADO NA INTEGRAÇÃO DE METODOLOGIAS ÁGEIS COM TRADICIONAIS.....	135
5 INTEGRAÇÃO DE FERRAMENTAS OPEN-SOURCE PARA ELABORAÇÃO DE INFRAESTRUTURA PARA IMPLANTAÇÃO CONTÍNUA.....	136
5.1 METODOLOGIA.....	136
5.1.1 Ambiente de Desenvolvimento Java	137
5.1.2 Máquina Virtual para Hospedagem da Infraestrutura	138
5.1.3 Configuração da Ferramenta de Containerização.....	139
5.1.3 Configuração do Servidor de Integração	140
5.1.4 Configuração do Repositório de Código e Controle de Versão	142
5.1.5 Configuração da Ferramenta de <i>Build</i> de Projetos	143
5.1.5 Configuração da Ferramenta de Testes Unitários.....	144
5.1.6 Configuração da Ferramenta de Testes Funcionais	144
5.1.7 Configuração da Ferramenta de Análise de Qualidade de Código .	147
5.1.8 Configuração da Ferramenta de Repositório de Artefatos.....	148
5.1.9 Criação de Contêiner Java para Execução do Projeto.....	150
5.1.10 Criação da <i>Pipeline</i> de Implantação Contínua.....	151
5.1.11 Tarefa para Execução do Arquivo de <i>Pipeline</i> e Blue Ocean	154
5.2 RESULTADOS OBTIDOS.....	155
5.2.1 Validação da Infraestrutura em um Cenário de Sucesso na <i>Pipeline</i>	156
5.2.2 Validação da Infraestrutura em um Cenário de Erro na <i>Pipeline</i>	161
6 CONCLUSÃO	164
REFERÊNCIAS.....	166

1 INTRODUÇÃO

A metodologia ágil é uma das práticas de desenvolvimento mais populares do momento, sendo utilizada por grandes empresas do ramo como Google e Microsoft. O desenvolvimento ágil se destaca por ajudar a entregar no software dentro do prazo e orçamento, com todas as funcionalidades originalmente prometidas ao cliente através da utilização de práticas como a integração contínua (SHORE, 2008, tradução nossa).

A integração contínua se destaca por ajudar a atenuar diversos problemas nas etapas de desenvolvimento, como a integração de uma parte do código ao resto do software ou casos onde o cliente presencia um problema que não é possível de ser reproduzido pela equipe, por exemplo, através da automatização de etapas de *builds*¹, testes e análises (KAWALEROWICZ; BERNTSON, 2011, tradução nossa).

Porém, mesmo apesar da grande difusão da metodologia ágil e suas práticas, como a integração contínua, as equipes de desenvolvimento ainda buscam conseguir um processo que seja previsível e de fácil reprodução, permitindo a implantação em produção de forma fácil e contínua (GAUDIN, 2015, tradução nossa).

De certa forma é necessário, para que as equipes de desenvolvimento publiquem um software em ambiente de produção, sejam executadas diversas etapas de forma manual, como por exemplo gerar o *build* e montar o ambiente que vai receber o mesmo... E tudo isso ocorre de forma descentralizada, o que dificulta o controle e a solução de problemas em virtude da dificuldade de reproduzir eventuais erros. Algumas soluções gratuitas existem, porém, são limitadas e não atendem a uma infraestrutura de implantação contínua completa.

Com isso, mediante o estudo da metodologia ágil de software e consequentemente a estrutura de uma *pipeline*² de implantação contínua verifica-se a possibilidade modelar uma completa infraestrutura de

¹ Ação de empacotar o código escrito em um arquivo executável (KAWALEROWICZ; BERNTSON, 2011, tradução nossa).

² Termo que significa um conjunto de etapas ou tarefas encadeadas para alcançar um determinado objetivo (SMART, 2011, tradução nossa).

desenvolvimento com o uso de ferramentas *open-source*.

1.1 OBJETIVO GERAL

Empregar ferramentas *open-source* para elaborar uma infraestrutura de implantação contínua em um ambiente de desenvolvimento Java.

1.2 OBJETIVOS ESPECÍFICOS

O trabalho tem como objetivos específicos:

- a) estudar a metodologia ágil de desenvolvimento e suas práticas, integração e implantação contínua;
- b) aplicar as práticas de integração e implantação contínua em um ambiente de desenvolvimento;
- c) executar a integração de diversas ferramentas *open-source* com diferentes finalidades ao Jenkins visando obter uma infraestrutura que possibilite a implantação contínua;
- d) empregar o Jenkins na criação de pipelines que consigam entregar softwares de maneira automática e contínua;
- e) desenvolver um protótipo em Java para web visando validar o funcionamento de toda a infraestrutura em múltiplos cenários.

1.3 JUSTIFICATIVA

Em uma empresa que não se utilize de práticas contínuas da metodologia ágil é comum existirem atrasos na entrega do software devido ao processo de desenvolvimento ser lento, cheio de processos manuais e ciclos para resolução de problemas demorados (PHILIPS et al, 2015, tradução nossa).

A utilização de uma prática como a implantação contínua propicia diversos benefícios, como (WOLFF, 2017, tradução nossa):

- a) redução de esforço decorrente de processos manuais que são automatizados;

- b) descoberta de erros mais cedo permitindo sua rápida correção;
- c) maior *feedback* do cliente, pois o software final é entregue com mais frequência;
- d) fácil reprodução de toda a *pipeline* por esta ser automatizada, incluindo a etapa de instalações de softwares, permitindo a reprodução do mesmo ambiente toda vez que a *pipeline* é executada.

Porém, para que as práticas contínuas sejam implementadas com sucesso e tragam resultado é necessária uma equipe comprometida e motivada bem como um domínio das aplicações utilizadas e uma infraestrutura apropriada (SHAHIN; BABAR; ZHU, 2017, tradução nossa).

Mediante a importância da utilização da metodologia ágil em um processo de desenvolvimento de software nos dias de hoje, faz-se necessária uma infraestrutura completa que seja apropriada para possibilitar a utilização de práticas contínuas de forma automatizada e que cubra todas as etapas de uma *pipeline* de entrega de software.

Com isso, para que esta infraestrutura possa ser modelada e testada, foi escolhida para o ambiente de desenvolvimento a linguagem de programação Java, essa que é a mais popular à época da escrita do desenvolvimento deste trabalho e, portanto, se mostra a ideal para exemplificar a utilização da infraestrutura (TIOBE, 2017).

1.4 ESTRUTURA DO TRABALHO

O presente trabalho é composto por 6 capítulos, sendo o primeiro uma introdução para o mesmo, buscando no primeiro subcapítulo contextualizar o leitor com relação a metodologia ágil, suas práticas e o problema em mãos. Nos subcapítulos posteriores são demonstrados os objetivos e a justificativa.

No capítulo 2 é descrito o que é engenharia de software, introduzindo aos subcapítulos seguintes que abordam as metodologias ágeis de desenvolvimento mais utilizadas.

No capítulo 3 é abordada a prática ágil de integração contínua,

introduzindo as etapas existentes de uma *pipeline* que visa implementar a mesma, e nos subcapítulos seguintes, cada etapa é descrita elencando sua importância.

No capítulo 4 são apresentados alguns trabalhos correlatos que dão uma ideia da importância da utilização da metodologia ágil de desenvolvimento e o que as difere de metodologias tradicionais. Os trabalhos também demonstram benefícios observados na utilização de algumas ferramentas citadas no presente trabalho visando atingir práticas como a integração contínua.

O capítulo 5 demonstra como foram configurados os ambientes que propiciaram o desenvolvimento de código e modelagem da infraestrutura de implantação contínua, explicando a integração de cada ferramenta da *pipeline*. Ao fim do capítulo são apresentados os resultados obtidos, explorando situações de sucesso e entrega do software em ambiente de produção ou onde o software não passou nos testes.

Por fim, no capítulo 6, é apresentada a conclusão com relação ao objetivo geral. São também sugeridos trabalhos futuros, demonstrando vias que podem ser exploradas de forma a expandir a aplicação da infraestrutura exposta no presente trabalho.

2 ENGENHARIA DE SOFTWARE

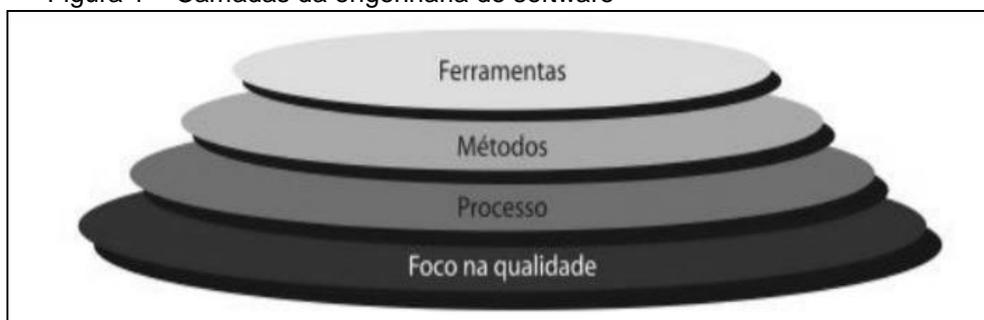
É necessário, para que se possa compreender sua engenharia, explicar o que é um software.

Software pode ser definido como qualquer programa executável escrito por profissionais do desenvolvimento para computadores de qualquer porte. Tem como função manipular um conjunto de informações de forma que um objetivo possa ser alcançado (PRESSMAN, 2011).

A engenharia de software pode ser definida como uma abordagem sistemática aplicada ao desenvolvimento de software para que este siga padrões aceitos pelo mercado de forma a manter um bom controle de qualidade. Tem como objetivo gerar softwares extremamente qualificados desde a operação até a manutenção (LEACH, 2016, tradução nossa).

A engenharia de software é uma tecnologia em camadas (figura 1), aonde há como base o foco da organização na qualidade do produto final como base de toda a abordagem. O processo, por sua vez, define como o projeto deve ser gerenciado, quais métodos devem ser aplicados e quais os objetivos. Métodos são utilizados para melhorar o ambiente de desenvolvimento através de uma boa comunicação da equipe, montagem da análise de requisitos e modelagem, escrita do código, etc. Por fim, as ferramentas são quem automatizam totalmente, ou quase, processos ou métodos (PRESSMAN, 2011).

Figura 1 – Camadas da engenharia de software



Fonte: Pressman (2011).

Existem aproximadamente sessenta metodologias de desenvolvimento de software, surgindo uma nova a cada dez meses, ao passo

de que ferramentas surgem duas a cada mês. Muitas já estão abandonadas há muito tempo por sua comprovada ineficácia, enquanto outras, como as metodologias ágeis, estão entre as mais utilizadas nos ambientes de desenvolvimento (JONES, 2017, tradução nossa).

2.1 METODOLOGIA ÁGIL DE DESENVOLVIMENTO

É de conhecimento comum que o desenvolvimento de software não é somente escrever código, envolvendo também as atividades de testes, *build* e *deploy*³ da aplicação final. Portanto, se faz necessária a adoção de uma metodologia que abranja as etapas do mesmo.

Uma metodologia é um conjunto de processos, técnicas e regras para se atingir um determinado objetivo (HOUAISS, 2009).

A metodologia ágil de desenvolvimento, por sua vez, é um conjunto de processos a serem seguidos visando o desenvolvimento de softwares de alta qualidade no menor tempo possível. Em relação a outras metodologias de desenvolvimento, ela se diferencia por trocar processos que geralmente acabam atrasando a entrega do software, como a documentação em demasia, por comunicação direta entre o time de desenvolvimento (GOODPASTURE, 2016, tradução nossa). Novos processos que agilizam a entrega também costumam ser adotados, como por exemplo a comunicação constante com os clientes durante o desenvolvimento para estar atento a mudanças em suas necessidades (SHORE, 2007, tradução nossa).

As metodologias ágeis atuais são pautadas no manifesto ágil, que é um embasamento filosófico escrito por profissionais de desenvolvimento em meados de 2001, o qual contém os valores e princípios que regem as práticas (GOMES, 2013).

³ Ação de entregar o software ao ambiente de execução destino (BERG, 2015, tradução nossa).

2.2 VALORES E PRÍNCÍPIOS

De acordo com Beedle et al. (2001), o manifesto ágil se divide em quatro valores: “Indivíduos e interações mais que processos e ferramentas; Software em funcionamento mais que documentação abrangente; Colaboração com o cliente mais que negociação de contratos; Responder a mudanças mais que seguir um plano”

Todos os valores do manifesto têm uma regra: os valores da direita são menos importantes que os valores a esquerda da sentença. O primeiro valor indica que se deve valorizar mais o relacionamento interpessoal entre o time de desenvolvimento do que os padrões e ferramentas a serem utilizados. O segundo valor diz que entregar o software é mais importante do que uma documentação em demasia. O terceiro valor sugere que o caminho a ser seguido durante o desenvolvimento é a colaboração com o cliente para estar atento a suas necessidades quanto ao software em detrimento da parte burocrática. Por fim o último valor ocorre quando através da colaboração do cliente, por exemplo, um feedback seja recebido e então o software seja prontamente alterado, ao invés de se seguir um plano predefinido imutável que conseqüentemente atrasará melhorias quanto as necessidades atuais do cliente (GOMES, 2013).

O manifesto ágil possui também doze princípios, uma extensão a seus quatro valores. Possui como princípio primordial e de maior prioridade a satisfação do cliente através da entrega de softwares o mais rápido possível. Os princípios funcionam visando atingir o princípio primordial, como ganhar vantagem no mercado respondendo rapidamente a mudanças, trabalhando em conjunto com as pessoas que solicitaram o desenvolvimento e entregando o software no menor tempo possível. Os demais princípios focam em aumentar a produtividade da equipe de desenvolvimento através de técnicas de motivação e conversas entre a mesma para redefinir e adotar novos conceitos visando conseguir uma maior atenção a excelência nos processos e auto-organização (APKE, 2014, tradução nossa).

2.3 MÉTODOS ÁGEIS

Nos últimos anos diversos métodos ágeis se popularizaram entre as equipes de desenvolvimento, isolados ou em conjunto, sendo Scrum, Lean, Kanban e XP os mais populares (LEFFINGWELL, 2011, tradução nossa).

O Scrum, amplamente adotado, se propõe a resolver problemas complexos através da melhora na habilidade de gerenciar prioridades. Isso ocorre com a utilização de um sistema de iterações, como por exemplo uma reunião mensal para reavaliar o produto desenvolvido até o momento e uma revisão geral ao final de cada dia (MAXIMINI, 2015, tradução nossa).

Lean, por sua vez, tem como base a ideia de evitar o desperdício visando adicionar mais valor ao produto final, isto é, não perder tempo desenvolvendo algo que não vá acrescentar valor ao produto no ponto de vista do cliente (POPPENDIECK, 2003, tradução nossa).

O método de gerenciamento de projetos Kanban impõe um sistema de visualização do fluxo de trabalho através de um quadro, dividindo as tarefas do projeto entre colunas que representam seu status (por exemplo, “não iniciado”, “em desenvolvimento”, “concluído”), sempre definindo um número mínimo e máximo de tarefas que podem ser feitas por etapas. Outro ponto importante do método é o cálculo de tempo médio estimado para completar determinada tarefa (LEOPOLD, 2015, tradução nossa).

Programação extrema, do inglês *eXtreme Programming* (XP), é um método ágil que possui diversas práticas que tem como objetivo permitir que o desenvolvimento tenha requisitos vagos ou mutáveis sem deixar de produzir um produto de qualidade. Porém, nem todas as práticas costumam ser adotadas pelas equipes devido a algumas controvérsias. Uma das práticas mais populares é a de integração contínua, que visa melhorar a parte de integração dos projetos (SATO, 2013).

3 INTEGRAÇÃO CONTÍNUA

No desenvolvimento de um projeto existe a fase de integração, onde são unificadas todas as modificações no código feitas pelos desenvolvedores, tendo então como resultado o software final. É uma etapa difícil, demorada e extremamente propensa a riscos pois uma vez que um conflito de código seja detectado pode levar muito tempo para verificar uma correção e suas implicações ao resto do projeto, o que acaba resultando em uma demora na entrega (SMART, 2011, tradução nossa).

Em virtude dos problemas que ocorrem durante a fase de integração de um projeto sem uma prática adequada proporcionada pela metodologia ágil surgiu a integração contínua, que se destaca por executar a integração do projeto de forma automática a cada *commit*⁴ de algum desenvolvedor para um repositório de código compartilhado, permitindo que um *build* seja gerado e erros sejam identificados e tratados em tempo de desenvolvimento, e não somente ao fim do projeto (FARCIC, 2016, tradução nossa). A responsabilidade principal da integração contínua é a automatização de testes unitários e *build* de projetos a partir de scripts, que pode ser estendida para cobrir testes específicos da aplicação, gerar relatórios de toda a *pipeline*, e até realizar o *deploy* visando atingir a implantação contínua (DAVIS, 2015, tradução nossa).

A utilização da integração contínua possui diversos benefícios, sendo eles: melhora na comunicação interna do time, ou até entre times em grandes projetos; maior produtividade dos desenvolvedores através das facilidades proporcionadas; e erros serem encontrados e até previstos mais facilmente, facilitando sua correção o mais rápido possível (STÅHL; BOSCH, 2013, tradução nossa).

Para que os benefícios de uma prática proveniente da metodologia ágil sejam atingidos é necessário, principalmente, existir um servidor de integração dedicado a tarefa e apenas uma fonte de repositório de código. É

⁴ Ação de gravar as alterações feitas no código em um repositório local (SOMASUNDARAM, 2013, tradução nossa).

preciso serem adotadas como regras de desenvolvimento por parte da equipe a alta mutabilidade do código sem nunca causar um erro no *build*, uma análise e testes prévios ao *commit* no próprio ambiente de desenvolvimento e a criação de um ambiente clone ao ambiente de produção também para testes. Por fim, deve-se buscar tornar o ambiente como um todo o mais automatizado e de fácil utilização para todos possível (WILDT et al., 2015).

O processo de scripts que automatizam os processos e integram ferramentas são chamados de *pipelines*. As *pipelines* devem rodar sempre que ocorrer um *push*⁵ para uma determinada *branch*⁶, e ao contrário da implantação contínua que possui como objetivo final a entrega em ambiente de produção, a integração contínua não possui um objetivo final predefinido, se focando em automatizar, basicamente, os testes e *build* das aplicações (FARCIC, 2016, tradução nossa).

3.1 INFRAESTRUTURA

Para que uma infraestrutura de integração contínua possa ser implementada são necessários que três requisitos principais sejam atendidos (HUMBLE; FARLEY, 2010, tradução nossa):

- a) controle de versão, onde todo o código desenvolvido deve estar centralizado em um repositório de código;
- b) *build* automatizado, onde precisa-se existir uma estrutura de empacotamento do projeto automatizada através de scripts, estes que devem ser tratados como qualquer outro código do projeto;
- c) comprometimento da equipe com a prática, no qual se dispõem a executar o *commit* várias vezes ao dia, prestar manutenção aos scripts de *build*, analisar os processos dos *build* automatizados visando identificar eventuais erros, e caso haja, corrigir o mais rápido possível.

⁵ Ação de enviar os *commits* feitos localmente para o repositório de código (SOMASUNDARAM, 2013, tradução nossa).

⁶ É um ramo de um projeto, geralmente utilizado para separar a escrita de diferentes módulos de um sistema (SOMASUNDARAM, 2013, tradução nossa).

Para implementar uma infraestrutura de integração contínua é preciso passar por diversas fases. Primeiramente é preciso adquirir alguma ferramenta que execute o *build* de sua aplicação conforme agendamento (geralmente feito em períodos de menor utilização, como durante a noite) e permita a análise do processo para que eventuais erros possam ser identificados e corrigidos. Posteriormente há a inclusão de testes automatizados na *pipeline* e a utilização de métricas para avaliar a qualidade de código automaticamente. Por fim, quando toda a estrutura esteja montada e seja confiável, pode-se analisar a possibilidade de estender o processo, visando, por exemplo, atingir a implantação contínua (SMART, 2011, tradução nossa). Uma infraestrutura de implantação contínua completa costuma ter ainda uma etapa de análise de qualidade do código e alguma ferramenta para armazenamento dos artefatos gerados durante o processo de *build* (WOLFF, 2017, tradução nossa).

3.2 IMPLANTAÇÃO CONTÍNUA

Como extensão a integração contínua existe a implantação (ou entrega) contínua.

Em um projeto que não se utilize deste processo há várias etapas executadas de forma manual que fazem com que a disponibilização de uma nova versão acabe demorando muito mais justamente para evitar essa fase, que é lenta, cheia de riscos e extremamente propensa a erros (WOLFF, 2017, tradução nossa).

Implantação contínua é o processo que automatiza a entrega de um software testado e funcional para um ambiente de produção a cada pequena mudança em alguma funcionalidade do sistema (SWARTOUT, 2012, tradução nossa). A *pipeline* da implantação contínua funciona como uma extensão a *pipeline* de integração contínua, onde a única intervenção do desenvolvedor é o *commit* para repositório de código, este que envia um sinal ao servidor responsável por integrar diversas ferramentas e automatizar todo este processo através de scripts, e termina com a aplicação executando em ambiente de produção (FARCIC, 2016, tradução nossa).

Uma prática comum é a utilização de autorização para implantação em um ambiente de produção, antes entregando a aplicação a um ambiente de homologação, este clone ao de produção, onde ela será acessada e testada (podendo ser de forma automática ou manual). Somente após os testes e autorização a aplicação é então liberada para o público geral (FARCIC, 2016, tradução nossa).

3.3 CONTROLE DE VERSÃO

É de conhecimento comum que para que haja um melhor controle do código escrito em um ambiente de desenvolvimento é necessário gerenciar e centralizar tudo em um repositório que proverá uma versão para cada desenvolvedor utilizar, bem como controlará situações de conflito de código.

Visando solucionar este problema foi criado o sistema de controle de versão, do inglês *Version Control System* (VCS). Em sua primeira geração, que possuía apenas as ferramentas RCS e SCCS, o desenvolvimento simultâneo de código era controlado através de “travas”, que permitiam somente uma pessoa editando ao mesmo tempo. Porém, devido a não permitir o desenvolvimento simultâneo, foi necessária uma melhoria significativa nas ferramentas, e assim surgiu a segunda geração. Nessa geração teve-se a criação das ferramentas Subversion, CVS, dentre outras. Foi nesse momento que as mesmas passaram a possibilitar desenvolvimento simultâneo sem travas, com a ressalva de que, para poder executar o *commit* de seu código, era necessário antes fazer um *merge*⁷ de tudo que foi feito por outros desenvolvedores no seu código. Por fim, a terceira geração de ferramentas nos trouxe o Git, Mercurial e Bazaar, permitindo que as edições pudessem ter seu *commit* feito sem ser necessário um *merge* (SINK, 2011, tradução nossa).

Uma ferramenta que se mostra referência, em grande parte por ter sido desenvolvida em cooperação com o criador do Linux e por ser utilizada com sucesso para gerenciar o código do mesmo sistema operacional (que possui

⁷ Ação de unir duas *branches* diferentes de desenvolvimento em uma única (SOMASUNDARAM, 2013, tradução nossa).

aproximadamente 9 milhões de linhas de conteúdo), é o Git. Como fatores principais para a escolha da ferramenta citada como solução deve-se citar (SOMASUNDARAM, 2013, tradução nossa):

- a) capacidade de evitar corrupção dos dados, pois não existe um meio termo na transação de um *commit*, ou ele é feito totalmente ou a versão anterior é mantida;
- b) desempenho comprovadamente rápido nas operações, confirmado pelos gigantes projetos que utilizam o sistema, como o já citado Linux;
- c) segurança como grande foco, utilizando-se de um *hash* SHA-1 para todos os arquivos, o que significa que não existe a possibilidade de alguém alterar o conteúdo dos mesmo sem o Git ficar sabendo.

A utilização de uma ferramenta de controle de versão possibilita um *rollback* a uma versão anterior do código com grande facilidade, pois guarda cada alteração no código na forma de *commit*, este que costuma conter também o autor responsável (DEMAREE, 2016, tradução nossa).

Apesar de todo o código ficar centralizado em um único lugar, cada desenvolvedor tem uma cópia local de todo o projeto, tornando todo o processo posterior de sincronização com o repositório central mais rápido. Além disso é possível separar um projeto em várias árvores de desenvolvimento, chamadas *branches*, permitindo que sejam escritos códigos para diversos cenários diferentes (NAREBSKI, 2016, tradução nossa).

Para facilitar a utilização de um sistema de controle de versão, como o Git, geralmente é utilizada alguma solução de gerenciamento de código fonte, as quais ajudam no sentido de que facilitam a utilização do mesmo, como por exemplo, implementando uma interface web para melhor administração dos projetos. Atualmente existe alguma solução na web servidas pelas próprias empresas das soluções, como Github, Bitbucket e Gitlab, este último que por sua vez permite a sua utilização local em uma infraestrutura própria (NAREBSKI, 2016, tradução nossa).

3.4 TESTES

Para que a qualidade do código seja garantida antes da entrega do software é de extrema importância a utilização de testes. A utilização de testes é importante pois ajuda a estabelecer confiança no código escrito, evitando que ocorram situações onde o software chega ao cliente com defeito, muitas vezes resultando em prejuízo e piora na reputação de quem desenvolveu. Mais importante do que possuir testes para o código, é executá-los regularmente, que é o que de fato dará confiança no que foi escrito (DUVALL; MATYAS; GLOVER, 2007, tradução nossa).

3.4.1 Testes Unitários

Testes unitários são testes escritos visando cobrir toda funcionalidade do software, e servem para adicionar uma camada de proteção ao desenvolvimento, facilitando a descoberta de *bugs* em situação atípicas que por natureza são difíceis de serem previstas e evitar que acabem chegando ao produto final do consumidor, causando insatisfação e eventual perda de tempo por parte da equipe visando corrigir o problema (APPEL, 2015, tradução nossa). A utilização de testes unitários garante que erros ou *bugs* no código possam ser encontrados em tempo de desenvolvimento, o que permite sua rápida correção. Isso dá segurança para colocar o projeto em uma *pipeline* de implantação contínua, possibilitando a entrega constante ao cliente com alta qualidade (SWARTOUT, 2012, tradução nossa).

Existe ainda a possibilidade de se escrever os testes unitários antes da funcionalidade que os mesmos irão validar, que é um conceito chamado de desenvolvimento orientado a testes. Basicamente define-se qual deverá ser o comportamento padrão daquela funcionalidade, o qual deverá ser descrito em forma de teste unitário. Desta forma quando a funcionalidade for escrita ela que deverá atender ao comportamento definido no código do teste escrito anteriormente (APPEL, 2015, tradução nossa).

Para que os testes escritos possam ser executados é necessária a utilização de alguma ferramenta de testes. Em caso de softwares desenvolvidos em Java há o JUnit e o TestNG (WOLFF, 2017, tradução nossa).

3.4.2 Testes Funcionais

Testes funcionais, também chamados de testes de aceitação, são testes que executam do ponto de vista do usuário final, isto é, não tem qualquer conhecimento interno de como o mesmo funciona (PERCIVAL, 2017, tradução nossa).

Para execução de testes funcionais em aplicações web há o Selenium, que possui um conjunto de ferramentas que permitem aos desenvolvedores “conversar” com o navegador (COLLIN, 2015, tradução nossa).

Como distinção quanto a testes unitários, é preciso dizer que estes testam a aplicação de um ponto de vista interno, focando em ajudar o desenvolvedor a escrever códigos limpos e livre de *bugs*. Os testes funcionais, por sua vez, são escritos descrevendo a funcionalidade do ponto de vista do usuário final, visando garantir que o software não vai parar de funcionar ao executar determinada funcionalidade de diversas formas possíveis (PERCIVAL, 2017, tradução nossa).

3.6 ANÁLISE DE QUALIDADE DE CÓDIGO

Em um projeto, devido a interação entre múltiplos desenvolvedores e seus códigos desenvolvidos, se faz necessário seguir à risca convenções de código para que o que for escrito possa ser entendido rapidamente por todos.

Caso o projeto não siga convenções de código sua leitura irá se tornar de difícil compreensão o que leva o tempo para que mudanças e novas funcionalidades possam ser adicionadas ao software a aumentar. A qualidade de código pode ser mensurada de diversas formas, como por exemplo verificar quantas possibilidades de execução uma classe possui ou então quão complexo é o código para a funcionalidade desejada (WOLFF, 2017, tradução nossa).

A equipe desenvolvedora do software de análise contínua de qualidade de código SonarQube define que para o código ter uma boa qualidade é preciso respeitar os padrões de código e focar em eliminar códigos propensos a *bugs*, duplicados e complexos. Outros fatores importantes para a análise de qualidade é o quão coberto por testes ou comentários e menos dependente de bibliotecas externas o código está (ARAPIDIS, 2012, tradução nossa).

Existem atualmente diversas ferramentas para analisar a qualidade do código, sendo algumas delas Checkstyle, FindBugs, PMD, Emma, etc., aonde sua utilização conjunta pode proporcionar uma análise completa do código. Como alternativa para a linguagem Java à utilização de todas as ferramentas citadas acima o software SonarQube, que possui diversas melhorias de qualidade de vida na parte de integração e apresentação de resultados ao usuário, inclusive integrando algumas ferramentas citadas acima (WOLFF, 2017, tradução nossa).

3.7 BUILD AUTOMATIZADO

Para que o software possa ser entregue ao usuário final o código escrito precisa ter feito seu *build*. Um *build* de uma aplicação consiste em compilar o código fonte, rodar os testes unitários definidos, processar os arquivos e por fim, gerar um artefato em um formato que possa ser lido por um sistema. Todo esse processo pode ter, ainda, a adição de algumas etapas, como uma análise da qualidade de código ou salvar o artefato gerado em um repositório centralizado (WOLFF, 2017, tradução nossa).

Em um processo de *build* manual se tem o problema do mesmo acabar sendo executado de forma distribuída por diversos desenvolvedores, o que o torna muito propenso a erros humanos, que acabam atrasando a entrega do produto final (MOODIE, 2005, tradução nossa). Para que os desenvolvedores possam se concentrar em desenvolver o código da aplicação final e evitar a perda de tempo proveniente do empacotamento do executável final de forma manual a utilização de um sistema de *build* automatizado se mostra indispensável (MITRA, 2015, tradução nossa).

A linguagem de desenvolvimento Java possui três ferramentas principais para a automatização de *build*, sendo elas (VARANASI, 2015, tradução nossa):

- a) Ant, lançado em 2000, se estrutura na forma de tarefas a serem completadas definidas em um arquivo .xml;
- b) Maven, lançado em 2004, se mostra a ferramenta mais popular para automatização de *builds* para softwares desenvolvidos em Java, e também se estrutura em .xml, com a diferença de que também adiciona o gerenciamento e declaração de dependências (bibliotecas externas) no seu arquivo .xml;
- c) Gradle, lançado em 2009 na sua primeira versão e em 2014 na segunda, é adição mais recente as ferramentas de automatização de *build*, e foi criado visando superar os problemas das duas ferramentas anteriores com base na experiência que as mesmas proporcionaram. Utiliza a linguagem Groovy para declarar seus scripts, uma técnica de *build* incremental que aumenta a velocidade dos processos, e declaração de dependências externas tal qual o Maven.

3.8 REPOSITÓRIO DE ARTEFATOS

Visando salvar os binários gerados pelos *builds* de uma *pipeline* de implantação contínua se faz necessária a utilização de um repositório de artefatos.

A prática de gerenciar um repositório de artefatos próprio se faz necessária pois ajuda no gerenciamento de dependências externas utilizadas nos projetos, garantindo que todos os projetos terão determinadas versões das mesmas disponíveis para utilização, ajudando na padronização evitando que em um projeto tenha-se uma dependência de versão mais antiga e em outro, uma versão mais nova. Outro ponto positivo é o controle de versões de *builds* já feitos para determinado software que foi desenvolvido e facilidade para auditoria de dependências utilizadas (HUMBLE; FARLEY, 2010, tradução nossa).

Empresas que prezam pela estabilidade devem sempre optar por um repositório de artefatos local em detrimento de um remoto. Isso pois, existindo um repositório local, há a confiança de que as dependências utilizadas não deixarão de estar disponíveis. É importante citar que os repositórios de artefatos locais geralmente possuem uma API REST, tecnologia esta que permitem que os mesmos sejam integrados a outras ferramentas (O'BRIEN et al., 2011, tradução nossa).

Existem duas ferramentas poderosas de uso livre, Nexus e Artifactory, que possuem como finalidade servirem como um repositório e gerenciador local de artefatos, seja eles as dependências utilizadas no desenvolvimento de um software (cache) ou os próprios softwares gerados (SMART, 2011, tradução nossa).

3.9 SERVIDOR DE INTEGRAÇÃO

A utilização de um servidor de integração contínua é recomendável caso se deseje a automatização de processos e relatórios dos mesmos, como por exemplo ficar verificando no repositório de controle versão se surgiu alguma modificação no código, e se surgiu, executar o *build* do projeto (DUVALL; MATYAS; GLOVER, 2007, tradução nossa).

Surgido a partir de um projeto do desenvolvedor Kohsuke Kawaguchi, o Hudson, foi um servidor de integração contínua que chegou a dominar 70% do mercado em meados de 2010. Porém, após a compra da Sun (empresa que prestava suporte ao desenvolvimento) pela Oracle, discussões com relação a como o projeto devia ser gerido começaram a surgir entre a comunidade e a nova empresa. Por fim a comunidade de desenvolvedores votaram para mudar o nome do projeto de Hudson para Jenkins e migrar todo o código para o site GitHub (SMART, 2011, tradução nossa).

Jenkins é um dos mais populares servidores de integração contínua do mercado, tendo sua popularidade amparada em sua fácil instalação em diferentes sistemas operacionais, interface simples e capacidade de ter suas funcionalidades estendidas através de *plugins* (SONI, 2015, tradução nossa).

Para que um servidor de integração contínua possa funcionar é necessária a utilização de *pipelines* definidas em script que executem as etapas sequenciais (ou em paralelo) que o mesmo deve executar para atingir o objetivo final. No Jenkins, o arquivo que armazena a *pipeline* é chamado de Jenkinsfile, podendo ser escrito em forma de script Groovy ou na sintaxe declarativa de *pipeline* (PATHANIA, 2017, tradução nossa).

A utilização de um servidor de integração contínua como o Jenkins, porém, ainda precisa uma correta manutenção, pois novos *plugins* e atualizações, que podem eventualmente possuir *bugs*, surgem diariamente. Há também o fato da necessária padronização dos scripts de cada projeto e as senhas de acesso que o servidor utiliza, essas que geralmente expiram e precisam ser trocadas (BERG, 2015, tradução nossa).

3.10 IMPLANTAÇÃO EM AMBIENTE DE PRODUÇÃO

Para que haja, por fim, a entrega do software em ambiente de produção, é necessária uma solução que tenha como foco a distribuição de software e possibilite que novos ambientes subam de forma rápida e simples.

O processo de distribuição do software pode ser feito através da utilização de máquinas virtuais, porém essa abordagem acaba gastando muitos recursos. Como alternativa existe a utilização da solução Docker que permite o emprego de contêineres Linux, estes que por sua vez empregam somente o *core* do *kernel* do Linux, tornando a sua implementação extremamente leve em comparação a utilização de máquinas virtuais (WOLFF, 2017, tradução nossa).

Outro benefício da utilização de contêineres Docker é que sua portabilidade permite eliminar *bugs* relativos a diferenças nos ambientes de execução das aplicações, pois sempre que um novo contêiner focado em aplicações é instanciado ele é idêntico a outro contêiner que possui a mesma finalidade (MOUAT, 2015, tradução nossa).

Atualmente o Docker é suportado pela maioria das empresas que prestam serviços de armazenamento na nuvem, como Google, IBM, Microsoft,

Amazon, dentre outros, o que cria confiança na adoção da ferramenta (MATTHIAS; KANE, 2015, tradução nossa).

4 TRABALHOS CORRELATOS

Este capítulo tem por objetivo relacionar os trabalhos que abordam a utilização da metodologia ágil com foco na integração contínua e suas extensões para melhorar o ambiente de desenvolvimento de softwares de forma semelhante a este trabalho.

4.1 IMPLANTAÇÃO DA INTEGRAÇÃO CONTÍNUA E SEUS BENEFÍCIOS: UM ESTUDO DE CASO

O presente trabalho de pesquisa foi elaborado por Daniel Baldo em janeiro de 2012, na Universidade do Vale do Rio dos Sinos, que se localiza na cidade de São Leopoldo, Rio Grande do Sul.

Essa monografia tem como objetivo demonstrar através de um estudo de caso a importância da utilização da integração contínua para que custos e riscos sejam minimizados.

A integração contínua foi aplicada com a ajuda de softwares a uma situação em que o setor de tecnologia da informação da empresa do autor realizava o desenvolvimento de projetos em colaboração com três fábricas externas de diferentes cidades do estado, o que ocasionava diversos problemas de integração. Para avaliar os resultados obtidos foram questionadas vinte e quatro pessoas relacionadas com o desenvolvimento, as quais responderam dez questões.

O resultado do questionário, após análise, demonstrou que grande parte das pessoas considerou que a integração contínua ajudou a solucionar diversos problemas, como retrabalho, identificação rápida de problemas, e entregas demoradas em ambiente de produção. Vale citar também que a equipe de desenvolvimento considerou que os testes automatizados e a análise de qualidade de código ajudaram a passar mais segurança e ajudar na antecipação de problemas.

4.2 INTEGRAÇÃO CONTÍNUA UTILIZANDO JENKINS

O seguinte trabalho de pesquisa foi elaborado por Lucas Lelis Bezerra e Sônia Santana em janeiro de 2013, no Centro Universitário do Triângulo, que se localiza na cidade de Uberlândia, Minas Gerais.

Este artigo científico tem como objetivo demonstrar os benefícios percebidos após a implementação da ferramenta Jenkins para atingir a metodologia ágil de integração contínua em um ambiente de desenvolvimento.

Para que a metodologia fosse posta à prova, após a implementação do Jenkins, foram monitorados nove projetos distintos, os quais tiveram seus *builds* executados pela mesma.

Concluiu-se que a utilização da metodologia ágil de integração contínua com o auxílio do Jenkins traz produtividade ao desenvolvimento de projetos, uma vez que reduz problemas de integração e centraliza a tarefa de gerar o *build* a uma máquina dedicada ao processo.

4.3 ETAPAS DE METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE ÁGIL BASEADA EM PROGRAMAÇÃO EXTREMA

O seguinte trabalho de pesquisa foi elaborado por Rodrigo Rosa da Rocha em junho de 2008, na Universidade do Extremo Sul Catarinense, que se localiza na cidade de Criciúma, Santa Catarina.

Essa monografia tem como objetivo apresentar os benefícios da utilização da metodologia ágil chamada programação extrema, esta que inclui as práticas de integração contínua, no desenvolvimento de um software.

Para observar os benefícios foi desenvolvido o protótipo de um site com ambiente virtual para uma escola. Ao final do desenvolvimento ficou claro que a programação extrema tornou o projeto mais interativo, verificando-se que torna o projeto mais interativo ao mesmo tempo em que reduz custos com relação a mudanças durante o projeto, além de demonstrar que as chances de o projeto final estarem de acordo com as expectativas do cliente são maiores do que na utilização de metodologias tradicionais.

4.4 ELABORAÇÃO DE UM FRAMEWORK PARA INTEGRAÇÃO DE SOFTWARE

O seguinte trabalho de pesquisa foi elaborado por Juliana da Silva Zafalão Tavares em dezembro de 2012, no Centro Universitário Eurípedes de Marília, que se localiza na cidade de Marília, São Paulo.

Essa monografia tem como objetivo criar um *framework* de desenvolvimento de software seguindo os princípios da integração contínua visando demonstrar os benefícios que podem ser conseguidos.

Ao final do trabalho foi observado que as utilizações das práticas de integração contínua são indispensáveis para um bom controle de projetos de desenvolvimento de software.

4.5 UTILIZAÇÃO DO FRAMEWORK EPF COMPOSER PARA A CRIAÇÃO DE UM MODELO DE PROCESSO DE DESENVOLVIMENTO DE SOFTWARE BASEADO NA INTEGRAÇÃO DE METODOLOGIAS ÁGEIS COM TRADICIONAIS

O seguinte trabalho de pesquisa foi elaborado por Ricardo Brina Mondardo em novembro de 2010, na Universidade do Extremo Sul Catarinense, que se localiza na cidade de Criciúma, Santa Catarina.

Essa monografia tem como objetivo criar um modelo de desenvolvimento de software que una as melhores práticas das metodologias tradicionais e ágeis.

Foi feito um estudo das principais metodologias de desenvolvimento e uma análise de ferramentas para integração contínua, sendo observado ao final do trabalho que a criação de um modelo misto utilizando as práticas ágeis e as tradicionais poderia ser satisfatória para utilização em projetos de até médio porte.

5 INTEGRAÇÃO DE FERRAMENTAS *OPEN-SOURCE* PARA ELABORAÇÃO DE INFRAESTRUTURA PARA IMPLANTAÇÃO CONTÍNUA

Este trabalho aqui proposto enfatiza a elaboração de uma infraestrutura para implantação contínua com foco em ambientes de desenvolvimento Java através da integração de ferramentas *open-source* utilizando como base a ferramenta de integração contínua Jenkins.

Tal infraestrutura, no âmbito da engenharia de software, proporciona a automatização de toda a *pipeline* de desenvolvimento, cobrindo desde o versionamento de código em um repositório até a entrega do software em ambiente de produção para o cliente final. De forma prática permite que os desenvolvedores se foquem inteiramente na escrita do código de novas funcionalidades que agreguem valor ao software o que, por consequência, acaba reduzindo o tempo necessário para a entrega de novas versões do software ao usuário final.

5.1 METODOLOGIA

Para que o trabalho pudesse ser executado foi montado um ambiente de desenvolvimento Java no computador local bem como uma infraestrutura em uma máquina virtual para implantação contínua.

Primeiramente foi feita a instalação do kit de desenvolvimento Java na máquina local e instalação da IDE IntelliJ IDEA. A seguir foi feita a instalação do software *open-source* Virtual Box, disponibilizado pela Oracle para virtualização de máquinas para que fosse possível a virtualização de uma distribuição do Linux Debian. Na máquina virtual em questão foi instalado o software *open-source* Docker para possibilitar a economia de recursos com o lançamento de contêineres para cada ferramenta a ser utilizada na *pipeline* de implantação contínua e aplicação desenvolvida a ser testada na mesma.

A etapa seguinte consistiu em instalar, configurar e integrar as ferramentas necessárias em contêineres Docker e organizar toda a *pipeline* de modo que a implantação contínua fosse possível.

Para que a infraestrutura pudesse ser utilizada foi necessário o desenvolvimento de algumas classes e arquivos de configuração em cima de um projeto Java para web já existente.

Por fim a infraestrutura de implantação contínua foi testada e observados cenários de erro (quando, por exemplo, o software não passou em algum teste) e de sucesso, nos quais o software foi colocado em ambiente de produção e entregue ao usuário final.

5.1.1 Ambiente de Desenvolvimento Java

A instalação do ambiente de desenvolvimento Java ocorreu com a utilização do kit de desenvolvimento disponibilizado pela Oracle em seu site oficial da versão 8 com build mais recente disponível no site da mesma⁸. Como interface de desenvolvimento foi escolhida a IDE IntelliJ IDEA, na qual foi utilizada a licença de estudante (vale lembrar, porém, que a escolha da IDE foi pessoal e não é algo obrigatório para a realização do trabalho, podendo ser utilizada qualquer outra IDE com suporte a Java para web).

A seguir foi escolhido o projeto em Java para web Spring Boot Petclinic⁹, um projeto *open-source* da fundação Pivotal, em virtude de suas atualizações constantes e completude quando comparado a um projeto comercial grande ele se mostra o ideal para teste da infraestrutura a ser desenvolvida. Tendo este projeto como base foram desenvolvidas classes de testes e arquivos de configuração visando uma *pipeline* de implantação contínua.

Vale lembrar também que não foi desenvolvido um projeto em Java para web do zero pois não é a finalidade deste trabalho, que propõe a utilização de uma aplicação web somente para teste da infraestrutura de implantação contínua. Porém ainda houve desenvolvimento por parte do autor, pois, como veremos nos capítulos seguintes, o mesmo precisa escrever testes de aceitação, cenários com o código correto ou errôneo e o arquivo denominado Jenkinsfile,

⁸ Disponível em: <http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html>

⁹ Disponível em: <https://github.com/spring-projects/spring-petclinic>

este que fica na raiz do projeto e define toda a *pipeline* de implantação contínua declarando que ferramentas devem ser utilizadas e as etapas pelas quais o processo é executado.

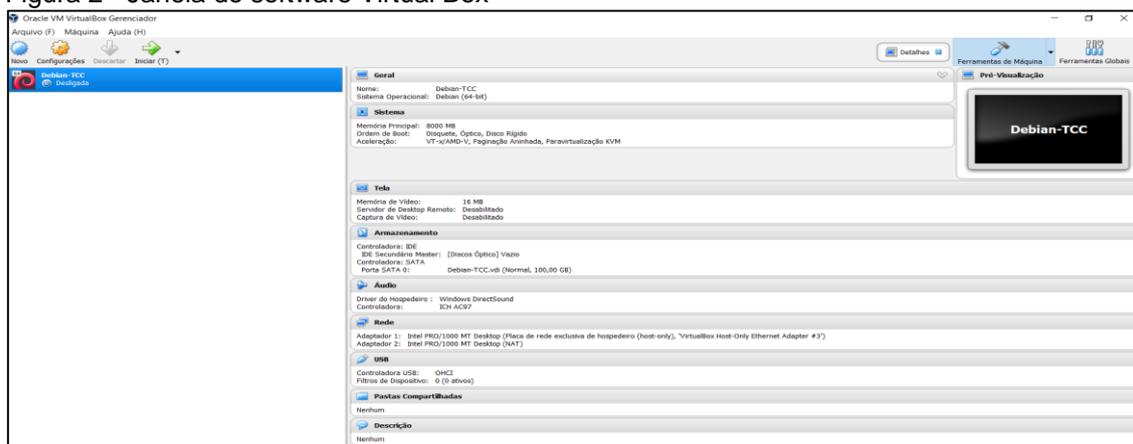
5.1.2 Máquina Virtual para Hospedagem da Infraestrutura

Para montagem da infraestrutura, visando a economia e melhor otimização dos recursos disponíveis foi optado pela utilização de uma única máquina virtual, a qual foi instalada com o software Virtual Box¹⁰. A distribuição de sistema operacional escolhida foi o Linux Debian¹¹ em sua versão 64bits, que é *open-source* e possui um ótimo suporte específico na documentação da ferramenta de containerização Docker.

Afim de criar uma nova máquina virtual o usuário deve clicar em *Novo* no canto superior esquerdo da janela do Virtual Box (figura 2), escolhendo então um nome e qual distribuição de sistema operacional será utilizada. Para registro a seguir estão as especificações que foram dadas para a máquina virtual nos passos seguintes:

- a) 8GB de memória RAM;
- b) 100GB de HD.

Figura 2 - Janela do software Virtual Box



Fonte: Do autor.

¹⁰ Disponível em: <https://www.virtualbox.org/wiki/Downloads>

¹¹ Disponível em: <https://www.debian.org/CD/http-ftp/#stable>

Por fim, para que a máquina virtual consiga acessar a internet e ao mesmo tempo seja visível por um IP estático localmente foram feitas algumas configurações. Primeiramente deve-se clicar no canto superior esquerdo em *Arquivo* e então *Host Network Manager*. A partir deste ponto deve-se clicar em *Criar* na janela que se abriu e então configurar os campos *Endereço IPv4* (este que será o IP estático visível para a máquina local que permitirá acessar a máquina virtual) e *Máscara de Rede IPv4* levando em consideração a realidade da rede local. A seguir deve-se efetuar duas configurações de rede clicando com o botão direito do mouse na máquina virtual e selecionando então *Configurações* e depois *Rede*. A primeira configuração de rede foi definida como *host-only*, e então selecionada em *Nome* a placa criada no passo anterior. A segunda configuração de rede, por sua vez, deve ser somente selecionada *NAT*. Salvando essas alterações a máquina virtual já estará acessível localmente pelo IP estático informada para a placa de rede criada acima e também terá acesso a internet para eventuais downloads de pacotes.

5.1.3 Configuração da Ferramenta de Containerização

A ferramenta de containerização Docker foi escolhida como alternativa para execução de cada ferramenta da *pipeline* de implantação contínua pois ela isola cada ferramenta com relação as outras em um sistema operacional dedicado ao mesmo tempo em que possibilita a economia de recursos, pois todos os contêineres compartilham os recursos da máquina virtual hospedeira.

Na figura 3 observa-se os passos para a instalação do Docker que devem ser executados na ordem determinada no terminal do Debian. Até o passo 6 é feita a configuração dos repositórios do Debian de onde será baixada a distribuição do Docker e no passo 7 é efetuada a instalação do software em questão.

Figura 3 - Código de Instalação do Docker

```
1 - sudo apt-get update
2 - sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg2 \
    software-properties-common
3 - curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
4 - sudo apt-key fingerprint 0EBFCD88
5 - sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/debian \
    $(lsb_release -cs) \
    stable"
6 - sudo apt-get update
7 - sudo apt-get install docker-ce
```

Fonte: Do autor.

5.1.3 Configuração do Servidor de Integração

É importante notar que a partir deste subcapítulo alguns comandos utilizados no terminal para execução dos contêineres não possuem o prefixo *sudo* pois foram executados com usuário *root* (geralmente o administrador em máquinas Linux).

Como solução para o servidor de integração foi utilizado o Jenkins, pois é uma alternativa *open-source* que possui um excelente suporte ao seu desenvolvimento, acumulando mais de dez mil estrelas de favorito em seu repositório principal (GITHUB, 2018), o que possibilita ao sistema ter uma natureza que facilite a integração a ferramentas externas.

Para instalação do Jenkins deve-se executar o comando vide figura 4 no terminal da máquina virtual, onde fica declarado de qual porta o mesmo será acessível pelo navegador, em qual pasta serão persistidos os dados, feita a exposição do *socket* do Docker (para que o Jenkins possa criar novos contêineres), definido o fuso horário no qual o mesmo irá rodar e passados parâmetros que definem o limite de utilização de memória do contêiner como sendo dois (2) gigabytes.

Figura 4 – Comando para execução do contêiner do Jenkins

```
docker run --restart always -dit -p 50000:8080 -p 50001:50000 --name jenkins \  
-e TZ=America/SaoPaulo --env JAVA_OPTS="-Xmx2048m -XX:MaxPermSize=512m" \  
-v /opt/docker-volumes/jenkins:/var/jenkins_home \  
-v /var/run/docker.sock:/var/run/docker.sock jenkins/jenkins:lts
```

Fonte: Do autor.

Após a execução do comando o Jenkins estará acessível na porta 50000 do endereço local definido na configuração feita no capítulo 5.1.2. Deve-se acessar este endereço para finalizar a instalação do sistema e, como será o primeiro acesso será pedida uma senha de administrador, a mesma pode ser adquirida utilizando o comando representado na figura 5 no terminal da máquina virtual.

Figura 5 – Comando para adquirir senha inicial de admin para o Jenkins

```
docker exec -it jenkins cat /var/jenkins_home/secrets/initialAdminPassword
```

Fonte: Do autor.

No prosseguimento da instalação deve-se confirmar a instalação dos *plugins* recomendados (mais utilizados pela comunidade) e efetuar o cadastro de um usuário e senha que serão as credenciais utilizadas em futuros acessos a ferramenta. Após o fim da instalação deve-se voltar ao terminal da máquina virtual e executar os comandos da figura 6 na ordem apresentada afim de instalar o Docker dentro do Jenkins, o que possibilitará ao sistema criar novos contêineres por demanda (como por exemplo os contêineres das aplicações Java que serão utilizados para validar essa infraestrutura).

Figura 6 – Instalação do Docker dentro do contêiner do Jenkins

```
1 - docker exec -it --user root jenkins /bin/bash  
2 - apt-get update && \  
apt-get -y install apt-transport-https \  
ca-certificates \  
curl \  
gnupg2 \  
software-properties-common && \  
curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")/gpg > /tmp/dkey; apt-key add /tmp/dkey && \  
add-apt-repository \  
"deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") \  
$(lsb_release -cs) \  
stable" && \  
apt-get update && \  
apt-get -y install docker-ce  
3 - usermod -a -G docker jenkins
```

Fonte: Do autor.

5.1.4 Configuração do Repositório de Código e Controle de Versão

Como já elencado anteriormente na fundamentação teórica do presente trabalho, o Git foi a ferramenta de controle de versão escolhida devido sua capacidade de evitar a corrupção de dados, performance superior e segurança como sendo um dos pilares de seu desenvolvimento.

Para facilitar o trabalho com a ferramenta deverá ser instalado o Gitlab, um sistema de gerenciamento de repositório, que irá prover uma interface para que sejam abstraídos comandos do Git que em caso contrário teriam de ser executados manualmente. A escolha do sistema em questão também levou em conta outro fator importante para a independência da infraestrutura de implantação, sendo ele a possibilidade de instalação local ainda mantendo seu perfil de utilização gratuita.

De forma a executar o sistema de gerenciamento de repositório Gitlab (que já possui o Git integrado) na estrutura que vem sendo seguida por este trabalho deve-se utilizar o comando conforme figura 7 no terminal da máquina virtual, onde é declarado as portas pelas quais o sistema estará acessível, fuso horário e pastas onde serão persistidos os arquivos.

Figura 7 - Comando de execução do Gitlab

```
docker run -dit --publish 50100:80 --hostname 192.168.76.5:50100 \  
  --publish 50101:443 --publish 50102:22 -e TZ=America/Sao_Paulo --name gitlab \  
  --restart always --volume /opt/docker-volumes/gitlab/config:/etc/gitlab:Z \  
  --volume /opt/docker-volumes/gitlab/logs:/var/log/gitlab:Z \  
  --volume /opt/docker-volumes/gitlab/data:/var/opt/gitlab:Z gitlab/gitlab-ce:latest
```

Fonte: Do autor.

Após a execução o sistema deverá estar acessível através do IP local configurado para a máquina virtual anteriormente na porta 50100. Ao acessar pela primeira vez será solicitado o cadastro da senha para o usuário *root*, e após confirmação o usuário deverá ser levado a tela de login do sistema.

Posteriormente, após efetuado *login* com as credenciais (usuário *root* e senha criada na etapa descrita acima) deverá ser criado um repositório para o projeto que será utilizado para validar a infraestrutura de implantação contínua.

Por fim copie o endereço HTTP do projeto recém criado no Gitlab e em sua interface de desenvolvimento de preferência, abra uma linha de comando e execute os comandos conforme figura 8. É importante lembrar que deve-se substituir **ENDEREÇO REPOSITÓRIO** pelo endereço HTTP copiado na etapa acima, que termina em .git e executar o comando na raiz do projeto.

Figura 8 – Comandos para sincronização do projeto a ser testado com o Gitlab

```
1 - git remote set-url origin *ENDEREÇO REPOSITÓRIO*  
2 - git push origin master
```

Fonte: Do autor.

Com isso o projeto que será usado para validar a infraestrutura de implantação contínua estará devidamente sincronizado ao seu repositório de código local, facilitando uma posterior integração ao servidor de integração. Por fim, para integrar as funcionalidades do Gitlab ao servidor de integração é necessário instalar o *plugin* do mesmo pelo caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*, então selecionando *Gitlab*.

5.1.5 Configuração da Ferramenta de *Build* de Projetos

Para que os projetos possam ser compilados em um executável é necessária uma ferramenta que execute o *build* dos projetos e nesta infraestrutura de implantação contínua foi escolhida dar suporte a utilização do Maven por conta de sua grande quantidade de bibliotecas disponíveis em seu repositório centralizado devido a sua popularidade, o que também acarreta em maior facilidade de integração ao Jenkins devido a *plugins* já existirem pela comunidade *open-source*.

Para instalar a ferramenta no Jenkins deve-se acessar o mesmo pelo endereço de IP local configurado previamente para a máquina virtual na porta 50000. Na tela inicial do sistema deve-se clicar em *Gerenciar Jenkins* no menu lateral e então em *Global Tools Configuration* na lista que é apresentada. A seguir deve-se procurar na lista de ferramentas a palavra *Maven*, e configurar o nome e versão de instalação conforme figura 9.

Figura 9 – Instalação da ferramenta Maven no Jenkins



The screenshot shows the 'Maven' configuration page in Jenkins. It features a 'Maven instalações' section with a 'Maven' icon and a 'Nome' input field containing 'Maven'. Below this is a checked checkbox for 'Instalar automaticamente'. At the bottom, there is an 'Instalar a partir do Apache' section with a 'Versão' dropdown menu set to '3.5.3'. A help icon is visible on the right side of the page.

Fonte: Do autor.

Por fim, para que a *pipeline* que será escrita no capítulo 5.1.10 deste trabalho possa ter acesso essa instalação um *plugin* precisará ser instalado acessando novamente *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*. Deve-se então procurar e marcar *Pipeline Maven Integration*, finalizando a instalação clicando em *Baixar, instalar e depois reiniciar* ao final da página.

5.1.5 Configuração da Ferramenta de Testes Unitários

A execução de testes unitários será feita com a utilização do JUnit, que possui uma excelente documentação em relação a concorrentes. Para que uma pipeline de implantação contínua possa executar testes unitários desta ferramenta será necessária a instalação de um *plugin* ao Jenkins, caso ainda não instalado, chamado *JUnit Plugin*, feita acessando o caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*.

5.1.6 Configuração da Ferramenta de Testes Funcionais

Para a tarefa de execução de testes funcionais, onde componentes do sistema web são testados simulando o comportamento humano, será utilizada a ferramenta Selenium, que é a alternativa para testes deste tipo em sistemas Java para web.

Aqui entra um empecilho com relação a utilização de contêineres: os mesmos não possuem telas, portanto o Jenkins não deverá conseguir executar os testes do Selenium. Como forma de contornar o problema será montada uma tela virtual, e para tal, deverá ser criada uma variável de ambiente no Jenkins. O usuário deverá fazer o caminho *Gerenciar Jenkins > Configurar o Sistema >*

Propriedades Globais e clicar em *Variáveis de Ambiente*, adicionando uma nova variável com nome *DISPLAY* e valor *:0*. Por fim o usuário deverá executar no terminal da máquina virtual o comando demonstrado na figura 10.

Figura 10 – Comando para criar uma tela virtualizada no contêiner do Jenkins

```
docker exec jenkins Xvfb :0 >& /dev/null &
```

Fonte: Do autor.

A execução de testes do Selenium irá ocorrer em um navegador, e para que os mesmos executem com sucesso será necessário instalar o Google Chrome dentro do contêiner do Jenkins, rodando os comandos no terminal da máquina virtual conforme figura 11.

Figura 11 – Comandos para instalação do Google Chrome no contêiner do Jenkins

```
1 - docker exec -it -u root jenkins /bin/bash
2 - curl -sS -o - https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add \
  echo "deb http://dl.google.com/linux/chrome/deb/ stable main" >> /etc/apt/sources.list.d/google-chrome.list \
  apt-get -y update \
  apt-get -y install google-chrome-stable \
  exit
```

Fonte: Do autor.

Será preciso, no projeto que será utilizado para validar a infraestrutura de implantação contínua, adicionar a dependência demonstrada na figura 12 ao arquivo *pom.xml* do projeto, para que o mesmo tenha acesso a biblioteca de testes.

Figura 12 - Dependência do Selenium

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>3.11.0</version>
</dependency>
```

Fonte: Do autor.

Deverá ser então escrita a classe Java que será o teste de aceitação da *pipeline*, a qual irá validar a possibilidade de clique no botão que possui o título *find owners* e caso não o encontre, irá diagnosticar um erro na aplicação. A classe em questão (figura 13) define o caminho em que o *driver* do Google

Chrome (este definido no parágrafo anterior) e, caso o botão exista, mostra no *log* da *pipeline* para qual endereço o usuário seria redirecionado ao clicar.

Figura 13 - Classe de teste funcional

```
@RunWith(SpringRunner.class)
@WebMvcTest(AcceptanceTests.class)
public class AcceptanceTests {

    private static String CHROME_DRIVER = "/chromedriver/chromedriver";

    @Test
    public void findButtonPetClinicTest() {
        File cDriver = new File(AcceptanceTests.class.getResource(CHROME_DRIVER).getFile());
        cDriver.setExecutable(true);
        System.setProperty("webdriver.chrome.driver", AcceptanceTests.class.getResource(CHROME_DRIVER).getFile());
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--headless", "window-size=1024,768", "--no-sandbox");
        ChromeDriver driver = new ChromeDriver(options);

        driver.get("http://192.168.76.5:60000");

        WebElement title = driver.findElement(By.xpath("//*[title='find owners']"));

        System.out.println("*****");
        System.out.println("*** ENDEREÇO URL *** " + title.getAttribute("href") + "*");
        System.out.println("*****");

        title.click();
        driver.quit();
    }
}
```

Fonte: Do autor.

Após a escrita da classe de testes será necessário baixar o *driver* do Google Chrome¹² para Linux dentro do projeto em *resources/chromedriver*, esta que é uma ferramenta auxiliar que possibilita que comandos de clique sejam dados pelo Selenium dentro do navegador.

Com a classe salva no repositório de código o servidor de integração já estará pronto para executar os testes de aceitação de que forem escritos (como o da figura 13).

¹² Disponível em: <http://chromedriver.chromium.org/downloads>

5.1.7 Configuração da Ferramenta de Análise de Qualidade de Código

De forma a assegurar que convenções de código sejam seguidas à risca em grandes equipes de desenvolvimento existem ferramentas que analisam a qualidade de código, como o Checkstyle ou ferramentas mais completas, como o Sonarqube, que permite a geração de relatórios e melhor análise através da atribuição de notas para cada projeto. Além disso esta última ferramenta disponibiliza um *plugin* de integração ao servidor de integração Jenkins chamado *SonarQube Scanner* que deve ser instalado através do caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*, e, portanto, se mostra a alternativa mais completa e de mais fácil integração dentre as disponíveis.

Para executar a ferramenta em um container seguindo o que vem sendo condicionado como estrutura no decorrer deste trabalho deve-se executar o comando conforme figura 14 no terminal da máquina virtual, o qual define as pastas em quais serão persistidos os dados e as portas por quais a mesma estará disponível. Após o fim da execução do comando a ferramenta deverá ter sua interface disponível para acesso através da porta 50300 no IP da máquina virtual, podendo ser acessada de qualquer navegador na máquina local.

Figura 14 – Comando para execução da ferramenta SonarQube

```
docker run -d --name sonarqube -p 50300:9000 -p 50301:9092
-v /opt/docker-volumes/sonarqube/data:/opt/sonarqube/data
-v /opt/docker-volumes/sonarqube/extensions:/opt/sonarqube/extensions sonarqube
```

Fonte: Do autor.

Na interface do sistema o usuário deverá ser apresentado a uma tela de *login*, a qual permitirá o acesso com as credenciais *admin* como nome de usuário e *admin* como senha. A primeira tela da ferramenta deverá apresentar uma solicitação de nome para o *token* de autenticação que será gerado para integrações a outras ferramentas, não existindo um nome ideal pré-definido, ficando então a cargo do usuário a escolha do mesmo. Após esta etapa o sistema informará uma sequência de caracteres (*token* de autenticação) os quais devem ser copiados para serem utilizados na etapa de integração ao Jenkins.

Para conectar a ferramenta ao servidor de integração deve-se efetuar a instalação do *plugin SonarQube Scanner* conforme já informado, e então acessar na interface do Jenkins sua configuração através do caminho *Gerenciar Jenkins > Global Tool Configuration > seção SonarQube Scanner*. Deve-se então clicando em *Adicionar SonarQube Scanner* e definir o *Name*, este que será utilizado para referenciar o *plugin* durante a *pipeline*.

Por fim a integração do SonarQube precisa ser configurada no Jenkins, o que pode ser feito acessando o caminho *Gerenciar Jenkins > Configurar o Sistema > seção SonarQube servers* (conforme figura 15). Nesta tela deve-se marcar a caixa *Enable injection of SonarQube server configuration as build environment variables*, informar um nome (campo *name*) e o endereço em que se encontra a ferramenta (campo *Server Url*), e informar a sequência de caracteres gerada na etapa anterior no campo *Server authentication token*. Com isso a ferramenta estará devidamente instalada e sua conexão com o servidor da integração configurada.

Figura 15 – Seção de configuração do SonarQube no Jenkins

SonarQube servers

Environment variables Enable injection of SonarQube server configuration as build environment variables
If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

SonarQube installations

Name

Server URL
Default is http://localhost:9000

Server authentication token
SonarQube authentication token. Mandatory when anonymous access is disabled.

Avançado...

Delete SonarQube

Fonte: Do autor.

5.1.8 Configuração da Ferramenta de Repositório de Artefatos

Ao final da *pipeline* de implantação contínua, logo antes da implantação em ambiente de produção, é necessário que o artefato gerado no processo seja arquivado para versionamento de versões implantadas com sucesso. Para resolver este problema existe a ferramenta Nexus, que é de

utilização livre e se propõe a arquivar tanto o *build* gerado como um todo como também as bibliotecas utilizadas, existindo a possibilidade de se dispensar a utilização do repositório central do Maven, por exemplo.

Para execução da ferramenta deve-se utilizar o comando demonstrado na figura 16 no terminal da máquina virtual, o qual declara as pastas em quais os dados serão persistidos e a porta na qual sua interface será exposta.

Figura 16 - Comando para execução da ferramenta Nexus

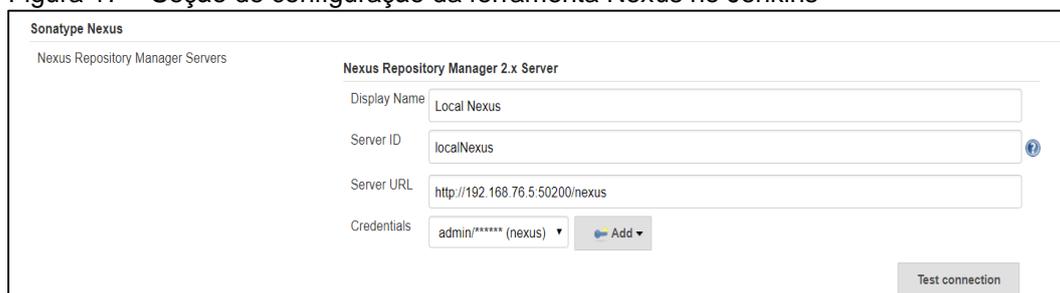
```
docker run -d -p 50200:8081 --name nexus  
-v /opt/docker-volumes/nexus:/sonatype-work sonatype/nexus
```

Fonte: Do autor.

Após a execução a ferramenta deverá ter sua interface disponível no IP local da máquina virtual na porta 50200 e poderá ser acessada usando *admin* como nome de usuário e *admin123* como senha.

Para efetuar a conexão da ferramenta ao servidor de integração deve-se instalar o *plugin Nexus Platform* através do caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*. Após a instalação do *plugin* deve-se seguir o caminho *Gerenciar Jenkins > Configurar o Sistema* e então procurar pela seção *Sonatype Nexus* (conforme figura 17),

Figura 17 – Seção de configuração da ferramenta Nexus no Jenkins



The screenshot shows the 'Sonatype Nexus' configuration page in Jenkins. The page title is 'Sonatype Nexus' and the subtitle is 'Nexus Repository Manager Servers'. The main heading is 'Nexus Repository Manager 2.x Server'. There are four input fields: 'Display Name' with the value 'Local Nexus', 'Server ID' with the value 'localNexus', 'Server URL' with the value 'http://192.168.76.5:50200/nexus', and 'Credentials' with a dropdown menu showing 'admin/***** (nexus)'. There is a blue 'Add' button next to the credentials dropdown. At the bottom right, there is a 'Test connection' button.

Fonte: Do autor.

Deve-se então informar valores a escolha do usuário para os campos *Nome de Exibição* e *DisplayID*. Para finalizar a configuração deve-se informar o endereço de IP da máquina virtual na porta exposta para a ferramenta Nexus em *Server URL* e as credenciais citadas no parágrafo acima nos respectivos

campos. Ao confirmar a configuração o Jenkins estará devidamente integrado ao repositório de artefatos Nexus.

5.1.9 Criação de Contêiner Java para Execução do Projeto

Para que a execução do projeto web possa ocorrer será necessário criar uma imagem de contêiner que suporte a execução de aplicações Java. Para isto será necessário criar um arquivo de nome *Dockerfile* dentro da máquina virtual que possui o Docker, este que permite a escrita de novos contêineres inclusive havendo a possibilidade de simplesmente adicionar novas funcionalidades a um já existente. Dentro do arquivo deverá existir três declarações (conforme figura 18):

- a) *FROM*: define que a imagem já existente *openjdk:8-jdk* será utilizada como base visando estender suas funcionalidades;
- b) *ENTRYPOINT*: define qual será o processo principal do contêiner, sendo então definido que será o executável Java disponível na pasta */opt/spring-app*;
- c) *RUN*: define um comando a ser rodado no momento do *build* da imagem, sendo então a criação da pasta */opt/spring-app* para onde será copiado o executável Java.

Figura 18 – Arquivo Dockerfile

```
FROM openjdk:8

ENTRYPOINT ["sh", "-c", "java -jar /opt/spring-app/app.jar"]

RUN mkdir -p /opt/spring-app/
```

Fonte: Do autor.

Por fim, para gerar a imagem de forma que ela possa ser executada futuramente para servir a aplicação web a ser utilizada nessa infraestrutura de implantação contínua deve-se chamar a funcionalidade de *build* do Docker no terminal da máquina virtual, definindo então o nome do mesmo como *java-app* (conforme figura 19).

Figura 19 – Comando para build do contêiner “java-app”

```
docker build -t "java-app" .
```

Fonte: Do autor.

5.1.10 Criação da *Pipeline* de Implantação Contínua

Com a instalação e configuração de todas as ferramentas necessárias para cada etapa da *pipeline* de implantação contínua é preciso, enfim, escrever o arquivo que descreve como a mesma será executada, o *Jenkinsfile*. Como já visto na fundamentação teórica o arquivo de *pipeline* possui dois formatos em que pode ser escrito, uma sendo a sintaxe declarativa e a outra sendo a sintaxe por script, sendo escolhida para execução deste trabalho a sintaxe declarativa, pois a mesma possui uma escrita mais clara e moderna com relação a sua alternativa.

Para começar será necessário criar um arquivo chamado *Jenkinsfile* na raiz do projeto Java para web escolhido para teste da *pipeline*. O arquivo deverá ter a declaração de uma chave chamada *pipeline* a qual englobará toda a estrutura do arquivo e então definir o *agent* (que é o servidor de integração Jenkins no qual o procedimento será executado) como *any*, já que na infraestrutura atual temos somente um Jenkins rodando. Para finalizar a estrutura inicial do arquivo (figura 20) é preciso também definir que será utilizada a ferramenta Maven durante as etapas utilizando a chave *tools*, sendo o nome o mesmo que foi definido na configuração da ferramenta de *build* nas configurações do sistema do Jenkins.

Figura 20 – Estrutura inicial do Jenkinsfile

```
pipeline {  
    agent any  
    tools {  
        maven 'Maven'  
    }  
}
```

Fonte: Do autor.

Para iniciar a declaração das etapas da *pipeline* de implantação contínua será necessário utilizar a chave *stages*, a qual conterá uma chave *stage* para cada etapa. Como primeira etapa a ser executada no processo foi definida a etapa de *build* automatizado e então implantação em um contêiner do projeto Java para web para que a etapa seguinte, que inclui testes funcionais, possa ser executada sem problemas. Nesta etapa são executados comandos que chamam a ferramenta Maven (já integrada a infraestrutura) com o comando *mvn* e efetivam o *build* sem execução de quaisquer testes, posteriormente implantando o mesmo em um contêiner que utiliza a imagem *java-app* (conforme figura 21), criada anteriormente.

Figura 21 – Estágio de build automatizado e implantação em contêiner para testes

```
stage('Build Automatizado e Implantacao em Container de Testes') {
  steps {
    echo '----> Iniciando compilacao do projeto em um container teste...'
    sh 'mvn package -DskipTests'
    sh 'docker create -p 60000:8090 --name java-test java-app'
    sh 'docker cp dist/app.jar java-test:/opt/spring-app/app.jar'
    sh 'docker network connect tcc-network java-test'
    sh 'docker start java-test'
    echo '----> Projeto para testes compilado e rodando com sucesso!!!'
  }
}
```

Fonte: Do autor.

A próxima etapa, como já citada, irá se tratar a definição da execução de testes unitários com JUnit e testes funcionais com Selenium. Para isso deverá ser adicionada mais uma chave *stage* (figura 22) dentro da estrutura já escrita, a qual conterá o comando de execução de testes *mvn test* (o qual irá chamar tanto o JUnit quanto o Selenium) e, em caso de sucesso dos testes, excluir o contêiner que foi criado exclusivamente para execução dos mesmos.

Figura 22 – Estágio para execução de testes unitários e funcionais

```
stage('Testes (Unitarios e Funcionais/Aceitacao)') {
  steps {
    echo '----> Iniciando o testes do projeto...'
    sh 'mvn test'
    echo '----> Projeto testado com sucesso!!!'
    echo '----> Deletando container para execucao de testes...'
    sh 'docker rm -f java-test || true'
    echo '----> Container para testes eliminado com sucesso!!!'
  }
}
```

Fonte: Do autor.

Em caso de sucesso na execução dos testes, conforme a *pipeline* de implantação contínua já definida, deve-se ocorrer a análise de qualidade de código, a qual irá fazer uma avaliação do código escrito e salvar o resultado atribuindo uma nota, que ficará registrada no Jenkins. O *stage* que será então escrito deverá definir a utilização do *SonarQube Scanner* já configurado anteriormente e executar o comando do Maven para *build* com análise de qualidade de código, conforme figura 23.

Figura 23 – Estágio para análise da qualidade de código

```
stage('Análise de Qualidade de Código - SonarQube') {
    steps {
        echo '---> Iniciando análise de qualidade de código...'
        script {
            scannerHome = tool 'SonarQube';
        }
        withSonarQubeEnv('SonarQube') {
            sh 'mvn package -DskipTests sonar:sonar'
        }
        echo '---> Análise de qualidade de código encerrada...'
    }
}
```

Fonte: Do autor.

Após a análise de qualidade de código e registro de seus resultados deve-se utilizar o repositório de artefatos Nexus para guardar o executável gerado pela *pipeline*, sendo então necessário adicionar um *stage* com o comando que chama o *plugin* do Nexus que salva o artefato conforme representado na figura 24.

Figura 24 – Estágio para salvar executável no repositório de artefatos Nexus

```
stage('Repositório de Artefatos - Nexus') {
    steps {
        echo '---> Iniciando o Upload do artefato gerado...'
        nexusPublisher nexusInstanceId: 'localNexus',
            nexusRepositoryId: 'releases', packages: [[class: 'MavenPackage',
            mavenAssetList: [[classifier: '', extension: '', filePath: 'dist/app.jar']],
            mavenCoordinate: [artifactId: 'spring-petclinic', groupId: 'org.springframework.samples',
            packaging: 'jar', version: '${BUILD_ID}']]
        echo '---> Upload do artefato executado e salvo com sucesso no repositório Nexus...'
    }
}
```

Fonte: Do autor.

Por fim, após salvar o executável no repositório de artefatos, existe a etapa de implantação em ambiente de produção visando a entrega da aplicação Java para web para utilização por parte dos usuários. O novo *stage* (figura 25) a ser adicionado deve efetuar a criação de um contêiner Docker que utilize a imagem *java-app* criada anteriormente, então copiando o executável Java gerado por esta *pipeline* e colocando dentro do contêiner.

Figura 25 – Estágio para implantação em ambiente de produção

```
stage('Implantacao em Ambiente de Produção') {
  steps {
    echo '---> Iniciando implantação em ambiente de produção...'
    sh 'docker create -p 8080:8090 --name java-app java-app'
    sh 'docker cp dist/app.jar java-app:/opt/spring-app/app.jar'
    sh 'docker start java-app'
    echo '---> Projeto implantado em ambiente de produção com sucesso!!!'
    echo '---> Está disponível na porta 8080 do host do Docker (http://192.168.76.5:8080)!!!'
  }
}
```

Fonte: Do autor.

Com a escrita dessas adições ao arquivo *Jenkinsfile* (que deverá ter efetuado seu *commit* para o repositório de código) toda a *pipeline* de implantação contínua proposta estará descrita em etapas (que são os *stages* definidos no arquivo) e o Jenkins estará apto a executar a mesma.

5.1.11 Tarefa para Execução do Arquivo de *Pipeline* e Blue Ocean

Para execução da *pipeline* escrita no capítulo 5.1.10 é necessário que seja criada uma tarefa (denominada *Job*) na interface do Jenkins clicando no botão *Novo Job* (acessível no menu lateral a esquerda), definindo então o nome da tarefa e selecionando a opção *Pipeline* na lista. Esta tarefa ficará encarregada de efetuar a integração entre o Jenkins e o repositório de controle de versão (Gitlab).

A configuração da tarefa deve ser feita de forma que na seção *Build Triggers* a opção *Build when a change is pushed to GitLab. GitLab webhook URL* seja marcada, guardando então o endereço que estará exposto logo ao lado da mesma. Tal configuração indicará que o Jenkins deve iniciar a *pipeline* automaticamente quando o Gitlab efetuar um *request* na URL exposta. Por fim, na seção *Pipeline* deve-se selecionar na lista *Pipeline from SCM* e então preencher o campo que indica a URL do repositório onde está hospedado o projeto e em *Script Path* nome do arquivo que contém a descrição da *pipeline* (no caso deste trabalho, o *Jenkinsfile* escrito no capítulo 5.1.10).

Uma última configuração na interface do Gitlab se faz necessária para que sempre que houver uma alteração no projeto que está sendo hospedado o mesmo efetue um *request* na URL exposta pela tarefa criada no Jenkins, sendo essa configuração que irá notificar o servidor de integração autorizando o início automático da *pipeline* de implantação contínua. Para isso deve-se acessar a página do projeto na interface do sistema de gerenciamento de controle de versão (Gitlab) e acessar a página de integrações através do caminho (disponível para acesso pelo menu lateral a esquerda) *Settings > Integrations*. Nesta página deve-se colocar no campo URL o endereço exposto pela tarefa criada no Jenkins e clicar na parte inferior da página em *Add webhook*.

Visando melhorar a apresentação da execução das *pipelines* quando forem demonstrados os resultados obtidos foi instalado o *plugin Blue Ocean* no Jenkins, este que é uma nova interface para melhor visualização da execução das *pipelines*, demonstrando claramente as etapas com um design moderno e de fácil entendimento. Para instalação do *plugin*, assim como outros que foram instalados ao longo deste trabalho, deve-se seguir o caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*, então selecionando *Blue Ocean* e confirmando a instalação.

5.2 RESULTADOS OBTIDOS

Durante o desenvolvimento do presente trabalho foi estudada a metodologia ágil de desenvolvimento e suas práticas. Esse estudo norteou a

definição de quais ferramentas deveriam ser utilizadas em cada etapa da *pipeline* final de implantação contínua, servindo como base para toda a aplicação prática deste trabalho de engenharia de software.

Pilar de uma infraestrutura de integração contínua a utilização de uma ferramenta de controle de versão foi adotada na escrita de código com o Git, onde cada alteração mínima tinha seu *commit* efetuado para o repositório de controle de versão (Gitlab), sincronizando o projeto o máximo possível.

Com a integração das ferramentas em um servidor de integração foi possível estruturar uma infraestrutura de implantação contínua com uma *pipeline* objetivando a entrega um projeto Java para web em ambiente de produção sem a interferência do programador (salvo, é claro, a parte de escrita dos códigos de testes e arquivo de *pipeline* em si).

5.2.1 Validação da Infraestrutura em um Cenário de Sucesso na *Pipeline*

Visando validar o funcionamento da infraestrutura em um cenário de sucesso (onde não há erros na *pipeline*) foi feita uma simples alteração em uma impressão no console na classe *AcceptanceTests* escrita no subcapítulo 5.1.6 (conforme figura 26) e efetuado seu *commit* para o repositório de código.

Figura 26 – Alteração em *AcceptanceTests* visando um cenário de sucesso na pipeline

```
33 System.out.println("*** TESTE DE SUCESSO - URL BOTAO ---> " + title.getAttribute("href") + "');
```

Fonte: Do autor.

De forma automática, ao ser efetuado o *commit*, o repositório de controle de versão (Gitlab) avisou ao servidor de integração (Jenkins) que o mesmo poderia iniciar a *pipeline* de implantação contínua conforme figura 27.

Figura 27 – Pipeline de implantação contínua iniciada pelo Gitlab

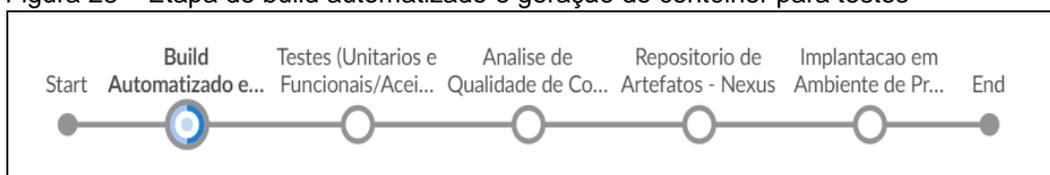
STATUS	RUN	COMMIT	MESSAGE
	57	–	Started by GitLab push by Administrator

Fonte: Do autor.

Tal feito possibilita que a interação do desenvolvedor com o processo de entrega do software seja praticamente nula, ficando como função do mesmo somente a escrita de código, o que propicia um maior foco na agregação de valor ao software em detrimento das intervenções manuais que seriam necessárias em uma outra infraestrutura que não siga os padrões de implantação contínua.

A *pipeline* então iniciará a primeira etapa que consiste no *build* automatizado com Maven e implantação do software Java para web em um contêiner Docker de testes (figura 28), que na etapa seguinte deverá ser utilizado para execução dos testes funcionais. Vale lembrar que para utilizar a interface da figura 28 que permite melhor visualização deve-se entrar na tarefa do Jenkins e clicar em *Open Blue Ocean* no menu lateral a esquerda.

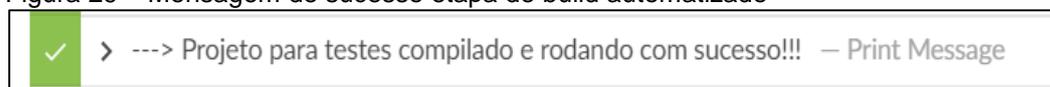
Figura 28 – Etapa de build automatizado e geração do contêiner para testes



Fonte: Do autor.

Ao fim desta etapa o *log* mostrará, conforme definido no arquivo Jenkinsfile anteriormente, uma mensagem indicando que a etapa foi completada com sucesso (figura 29).

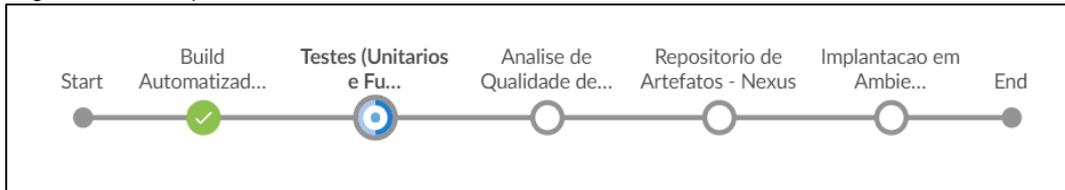
Figura 29 – Mensagem de sucesso etapa de build automatizado



Fonte: Do autor.

O Jenkins então iniciará a próxima etapa (figura 30), que consiste em executar os testes unitários com JUnit e funcionais com Selenium, e gravar seus resultados para consulta posterior. É importante notar que ao fim do processo o contêiner de testes criado anteriormente é encerrado visando a economia de recursos.

Figura 30 – Etapa de testes unitários e funcionais



Fonte: Do autor.

Nesta etapa será possível ver no *log* do comando *mvn test* a alteração na impressão efetuada no início do presente subcapítulo na classe *AcceptanceTests*, indicando que o botão *find owners* foi encontrado na página e é clicável (figura 31).

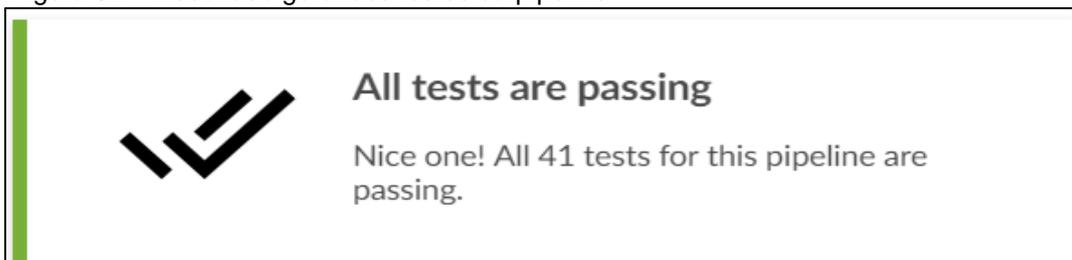
Figura 31 – Impressão no log de alteração efetuada na classe *AcceptanceTests*

```
310 *****
311 *** TESTE DE SUCESSO - URL BOTAO ---> http://192.168.76.5:60000/owners/find*
312 *****
```

Fonte: Do autor.

É importante ressaltar que se o usuário quiser simplesmente ter uma visão geral dos testes pode-se clicar no menu superior em *Tests*, o qual demonstrará uma lista de testes que foram executados e qual o resultado geral (figura 32).

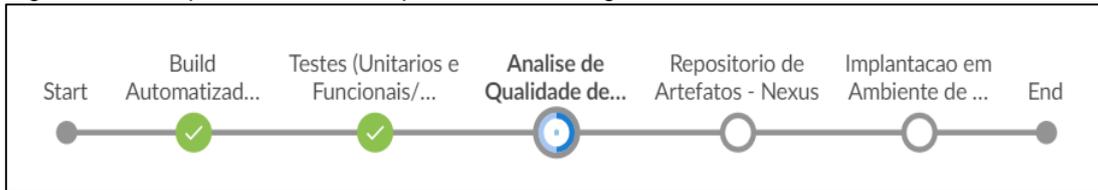
Figura 32 – Resultado geral dos testes da pipeline



Fonte: Do autor.

A etapa a seguir (figura 33) dá início a análise da qualidade de código com a ferramenta SonarQube, avaliando todo o código do projeto Java para web em comparação a convenções e melhores práticas.

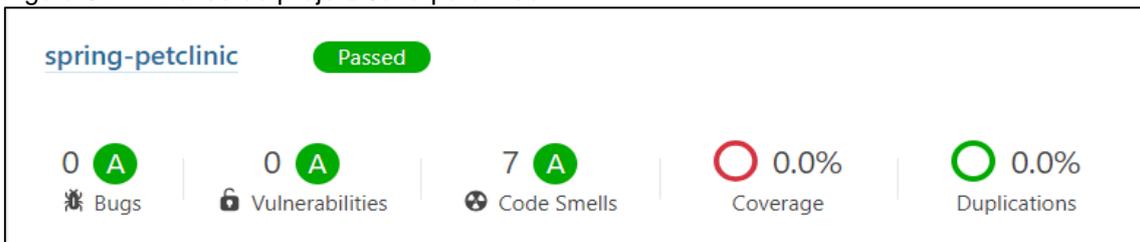
Figura 33 – Etapa de análise de qualidade de código



Fonte: Do autor.

Ao fim do processo de análise do código os resultados deverão ficar gravados na interface da ferramenta SonarQube, podendo então serem acessados através do IP da máquina virtual na porta 50300. Na página de projetos o usuário será confrontado com a avaliação geral (figura 34) que indica uma nota para vários fatores, dentre eles bugs no código, vulnerabilidades e porcentagem de código duplicado.

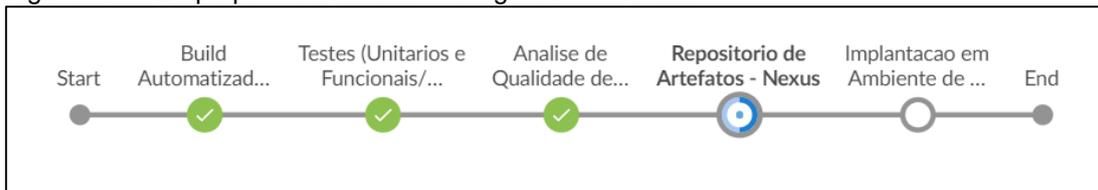
Figura 34 – Análise do projeto Java para web



Fonte: Do autor.

Com o *build* automatizado, testes e análise de qualidade de código efetuados com sucesso a *pipeline* então iniciará a etapa de salvar o artefato (executável Java) em um repositório externo da ferramenta Nexus (figura 35), visando conseguir um versionamento e arquivamento de *builds*.

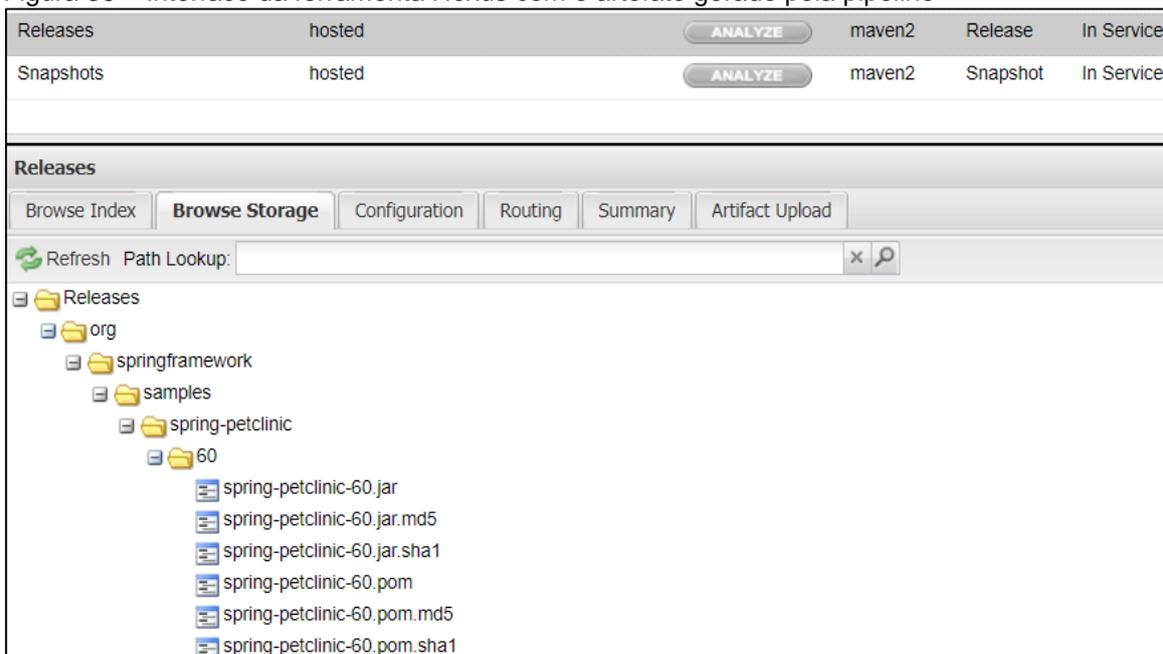
Figura 35 – Etapa para salvar o artefato gerado no Nexus



Fonte: Do autor.

Com a conclusão desta etapa o artefato poderá ser consultado na interface da ferramenta Nexus (figura 36) através do IP da máquina virtual na porta 50200, podendo então analisar o arquivamento de todas as versões já geradas, e caso necessário, utilizar as mesmas.

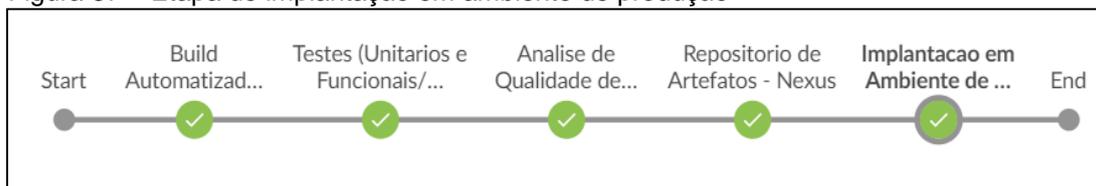
Figura 36 – Interface da ferramenta Nexus com o artefato gerado pela pipeline



Fonte: Do autor.

De volta a *pipeline* chega-se a etapa final da implantação contínua: a entrega do software em ambiente de produção então disponibilizando-o para o usuário final. Esta etapa (figura 37), conforme descrito no arquivo *Jenkinsfile*, consiste em pegar o *build* gerado pela *pipeline* e implantar em um contêiner pronto para execução de aplicações Java para web na porta 8080 (figura 38) da máquina virtual.

Figura 37 – Etapa de implantação em ambiente de produção



Fonte: Do autor.

Figura 38 – Implantação em ambiente de produção com sucesso

```
✓ > ---> Projeto implantado em ambiente de produção com sucesso!!! – Print Message
✓ > ---> Está disponível na porta 8080 do host do Docker (http://192.168.76.5:8080)!!! – Print Message
```

Fonte: Do autor.

Com a conclusão da etapa de implantação em ambiente de produção é finalizada também toda a *pipeline* de implantação contínua validando um caso sucesso, que como já explicado em demasia durante o decorrer deste trabalho, cobre desde o *build* automatizado até a entrega do software em ambiente de produção. O projeto Java para web então deverá estar disponível (figura 39) para acesso pelos usuários finais.

Figura 39 – Projeto Java para web entregue em ambiente de produção



Fonte: Do autor.

5.2.2 Validação da Infraestrutura em um Cenário de Erro na *Pipeline*

Visando validar a infraestrutura em um cenário de erro (onde há falhas na *pipeline*) foi feita uma alteração na classe *AcceptanceTests* escrita no subcapítulo 5.1.6 (conforme figura 40) e efetuado seu *commit* para o repositório de código. A alteração consiste em dizer para o teste funcional (que simula o comportamento humano verificando a existência de componentes no projeto Java para web) clicar no botão de título *dog* ao invés de *find owners*.

Figura 40 – Alteração em *AcceptanceTests* visando um cenário de erro na *pipeline*

```
35 WebElement title = driver.findElement(By.xpath("//*[@title='dog']"));
```

Fonte: Do autor.

Ao efetuar o *commit* o Gitlab irá avisar ao Jenkins que houve uma alteração no repositório de código para que o mesmo inicie a *pipeline* de implantação contínua, que irá começar com o *build* automatizado e implantação de um contêiner da aplicação Java para web visando a execução de testes funcionais que acontecerá na próxima etapa.

Porém, como há um erro (escrito propositalmente, pois um botão de título *dog* não existe na aplicação), uma falha na *pipeline* irá ocorrer durante a execução dos testes (figura 41) interrompendo a execução da mesma.

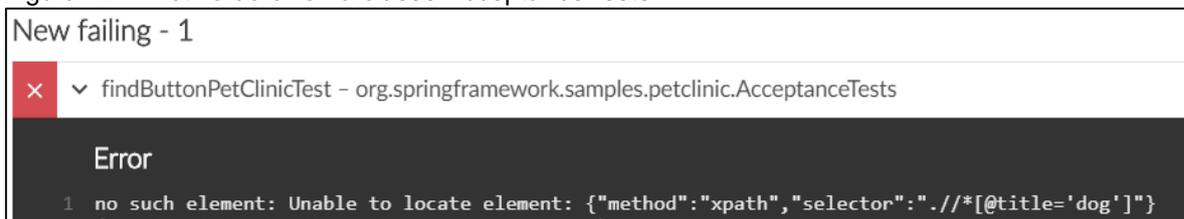
Figura 41 – Erro na etapa de testes unitários e funcionais



Fonte: Do autor.

Clicando em *Tests* no menu superior será possível consultar todos os testes que foram executados e em quais ocorreu um erro. Em caso de erros haverá uma lista de testes que falharam e no caso deste trabalho em específico será possível observar o erro na classe *AcceptanceTests* e o motivo (figura 42), sendo este a impossibilidade de localizar o botão com título *dog*.

Figura 42 – Motivo do erro na classe AcceptanceTests



Fonte: Do autor.

Com a interrupção e encerramento forçado da *pipeline* de implantação contínua devido a uma falha identificada por um teste funcional é possível validar um cenário de erro do processo, onde é feito um arquivamento dos erros que

ocorreram por parte do Jenkins para consulta posterior pelos desenvolvedores e o software nunca chega no ambiente de produção.

6 CONCLUSÃO

Para a configuração de uma infraestrutura que permitisse a prática da implantação contínua foi necessário o estudo das metodologias ágeis que englobam a mesma, sendo esse o que acabou por nortear a tomada de decisões durante todo o desenvolvimento do trabalho.

Na fase de desenvolvimento dos códigos para o projeto Java para web o estudo das práticas ágeis influenciou na adoção de uma abordagem em que se utiliza uma ferramenta de controle de versão para que alterações no código fiquem melhor documentadas ou até voltar o código ao estado de *commits* anteriores.

A fase de configuração da infraestrutura em si, isto é, a definição e organização das etapas de uma *pipeline* de implantação contínua e escolha da melhor forma para estruturar a mesma (sendo escolhida a utilização de uma máquina virtual com contêineres Docker) foi totalmente amparada nos conhecimentos adquiridos durante esse mesmo estudo.

Para finalizar a etapa de estruturação da base do presente trabalho foi feita a escolha e integração de ferramentas que atenderam a finalidade de cada etapa da *pipeline* de implantação contínua, o que mostrou um resultado positivo imediato devido ao grande suporte existente da comunidade *open-source* ao servidor de integração Jenkins.

Com o emprego do Jenkins (e as ferramentas integradas) na criação e execução de *pipelines* de implantação contínua foi possível automatizar todo o processo de entrega de software desde o *build* automatizado até que o mesmo chegasse ao ambiente de produção sendo então entregue ao usuário final.

É importante ressaltar também o aspecto de entrega contínua que a infraestrutura possibilitou, isto é, após uma pipeline estar configurada todos os *builds* posteriores irão apenas repetir um fluxo predefinido, o que transformou a dificuldade de se entregar o software devido a processos manuais e propensos a falha humana em algo inexistente.

O desenvolvimento do protótipo em Java para web, baseado em um projeto já existente e *open-source*, focou na escrita de novas classes e arquivos

de configuração necessários a execução da *pipeline* de implantação contínua. Ao final foi possível validar o funcionamento de toda a infraestrutura, podendo serem observados cenários de sucesso, onde o software foi entregue em ambiente de produção e cenários de falha, onde o mesmo não passou em alguma classe de teste escrita para cobrir erros no software, então documentando a causa do erro para consulta por parte dos desenvolvedores.

Uma observação importante diz respeito ao fato de que foi designado um foco em configurar uma infraestrutura que fosse agnóstica com relação a *frameworks* utilizados no desenvolvimento do projeto Java para web visando aumentar os cenários em que a mesma pudesse ser utilizada. Dada a forma com a qual a infraestrutura foi modelada é possível afirmar que a mesma pode atender a qualquer software desenvolvido em Java para web que utilize as mesmas ferramentas de *build* e testes deste trabalho, e em caso contrário, sendo necessárias somente pequenas alterações, como por exemplo a instalação de um *plugin* para a ferramenta divergente ao Jenkins.

Com isto, é possível afirmar que todos os objetivos específicos propostos para o presente trabalho foram alcançados, o que conseqüentemente proporcionou a elaboração de uma infraestrutura de implantação contínua plenamente funcional.

Propõe-se para trabalhos futuros a modelagem de uma infraestrutura de implantação contínua possuindo como foco softwares desktop, mobile e/ou outras linguagens de programação. Outra possibilidade de trabalho futuro consiste em uma análise com métricas que estimem o desempenho dos desenvolvedores utilizando-se de uma infraestrutura de implantação contínua em comparação a um cenário em que não haja a implantação automatizada. Por fim, fica a sugestão de um estudo que busque identificar através de métricas qual são as melhores ferramentas para cada etapa de uma *pipeline* de implantação contínua.

Pode-se dar continuidade a este trabalho modelando uma *pipeline* que possibilite a entrega de softwares que dependem de múltiplas implantações em produção, como por exemplo em um cenário em que exista um software servindo como API e diversos outros como interface para o usuário final.

REFERÊNCIAS

ARAPIDIS, Charalampos S.. **Sonar Code Quality Testing Essentials**. Birmingham: Packt Publishing, 2012. 318 p.

BEEDLE, Mike et al. **Manifesto para Desenvolvimento Ágil de Software**. 2001. Disponível em: <<http://agilemanifesto.org/iso/ptbr/manifesto.html>>. Acesso em: 13 ago. 2017.

BERG, Alan Mark. **Jenkins Continuous Integration Cookbook**: Over 90 recipes to produce great results from Jenkins using pro-level practices, techniques, and solutions. Birmingham: Packt Publishing, 2015. 360 p.

COLLIN, Mark **Mastering Selenium WebDriver**: Increase the performance, capability, and reliability of your automated checks by mastering Selenium WebDriver. Birmingham: Packt Publishing, 2015. 280 p.

DAVIS, Christopher W. H.. **Agile Metrics in Action**: How to measure and improve team performance. Greenwich, Connecticut: Manning Publications, 2015. 272 p.

DEMAREE, David. **Git For Humans**. Nova Iorque: A Book Apart, 2016. 134 p.

DUVALL, Paul M.; MATYAS, Steve; GLOVER, Andrew. **Continuous Integration**: Improving Software Quality and Reducing Risk. Boston: Addison-wesley Professional, 2007. 336 p.

FARCIC, Viktor. **The DevOps 2.0 Toolkit**. Birmingham: Packt Publishing, 2016. 430 p.

GAUDIN, Olivier. Prefácio. In: DAVIS, Christopher W. H.. **Agile Metrics in Action**. Greenwich, Connecticut: Manning Publications, 2015. 272 p.

GITHUB. Jenkins automation server. Disponível em: <<https://github.com/jenkinsci/jenkins>>. Acesso em: 01 de mai. 2018.

GOMES, André Faria. **Agile**: Desenvolvimento de software com entregas frequentes e foco no valor de negócio. São Paulo: Casa do Código, 2013. 163 p.

GOODPASTURE, John C.. **Project Management the Agile Way**: Making it Work in the Enterprise. 2. ed. S Pine Island Rd, Florida: J. Ross Publishing, 2016. 393 p.

HOUAISS, Antônio. **Dicionário Houaiss Da Língua Portuguesa - Novo**. Rio de Janeiro: Editora Objetiva Ltda., 2009. 2048 p.

HUMBLE, Jez; FARLEY, David. **Continuous Delivery**: reliable software releases through build, test, and deployment automation. Boston: Addison-wesley Professional, 2010. 512 p.

JONES, Capers **Software Methodologies**: a quantitative guide. 2. ed. Boca Raton, Florida: Auerbach Publications/CRC, 2017. 578 p.

KAWALEROWICZ, Marcin; BERNTSON, Craig. **Continuous Integration in .NET**. Greenwich, Connecticut: Manning Publications, 2011. 375 p.

LEACH, Ronald J. **Introduction to Software Engineering**. 2. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2016. 428 p.

LEFFINGWELL, Dean. **Agile Software Requirements**: Lean Requirements Practices for Teams, Programs, and the Enterprise. Boston: Addison-Wesley Professional, 2011. 560 p.

LEOPOLD, Klaus; KALTENECKER, Siegfried. **Kanban Change Leadership**: Creating a Culture of Continuous Improvement. New Jersey: Wiley, 2015. 320 p.

MATTHIAS, Karl; KANE, Sean P.. **Docker Up and Running**: Shipping Reliable Containers in Production. Sebastopol, Ca: O'reilly Media, 2015. 222 p.

MAXIMINI, Dominik. **The Scrum Culture**: Introducing Agile Methods in Organizations. New York: Springer International Publishing, 2015. 315 p.

MITRA, Mainak. **Mastering Gradle**: Master the technique of developing, migrating, and building automation using Gradle. Birmingham: Packt Publishing, 2015. 279 p.

MOODIE, Matthew. **Pro Apache Ant**. Nova Iorque: Apress, 2005. 342 p.

MOUAT, Adrian. **Using Docker**: Developing and Deploying Software with Containers. Sebastopol, Ca: O'reilly Media, 2015. 355 p.

NAREBSKI, Jakub. **Mastering Git**. Birmingham: Packt Publishing, 2016. 418 p.

O'BRIEN, et al. **Repository Management with Nexus**. 3. ed. Montgomery: Sonatype, 2011. 262 p.

PATHANIA, Nikhil. **Pro Continuous Delivery**: with Jenkins 2.0. Nova Iorque: Apress, 2017. 298 p.

PERCIVAL, Harry. **Test-Driven Development with Python**: Test-Driven Development with Python. 2. ed. Sebastopol, Ca: O'reilly Media, 2017. 614 p.

PHILLIPS, A. et al. **The IT Manager's Guide to Continuous Delivery**: Delivering business value in hours, not months. BookBaby, 2014. 54 p.

POPPENDIECK, Mary; POPPENDIECK, Tom. **Lean Software Development: Lean Requirements Practices for Teams, Programs, and the Enterprise**. Boston: Addison-Wesley Professional, 2003. 240 p.

PRESSMAN, Roger S. **Engenharia de Software: uma abordagem profissional**. 7. ed. Nova Iorque: Mc Graw Hill, 2011. 780 p.

SATO, Danilo. **DevOps na prática entrega de software confiável e automatizada**. São Paulo: Casa do Código, 2013. 284 p.

SHAHIN, Mojtaba; BABAR, Muhammad Ali; ZHU, Liming. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. **IEEE Access**, v. 5, 2017. 35 p. Disponível em: < <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7884954>>. Acesso em: 15 nov. 2017.

SHORE, James; WARDEN, Shane. **The Art of Agile Development**. Sebastopol, Ca: O'reilly Media, 2007. 432 p.

SMART, John Ferguson. **Jenkins: The Definitive Guide**. Sebastopol, Ca: O'reilly Media, 2011. 404 p.

SONI, Mitesh. **Jenkins Essentials: Continuous Integration – setting up the stage for a DevOps culture**. Birmingham: Packt Publishing, 2015. 186 p.

STÅHL, Daniel; BOSCH, Jan. IASTED INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 12., 2013, Innsbruck, Áustria. **Experienced benefits of continuous integration in industry software product development: A case study**. 8p. Disponível em: < https://www.researchgate.net/profile/Jan_Bosch/publication/266632251_Experienced_Benefits_of_Continuous_Integration_in_Industry_Software_Product_Development_A_Case_Study/links/54bc05a90cf253b50e2d149b.pdf>. Acesso em: 09 out. 2017.

SWARTOUT, Paul. **Continuous Delivery and DevOps: A Quickstart guide**. Birmingham: Packt Publishing, 2012. 154 p.

TIOBE. Index for November 2017. Disponível em: <<https://www.tiobe.com/tiobe-index>>. Acesso em: 15 de nov. 2017.

WILDT, Daniel et al. **EXtreme Programming: Práticas para o dia a dia no desenvolvimento ágil de software**. São Paulo: Casa do Código, 2015. 148 p.

WOLFF, Eberhard. **A Practical Guide to Continuous Delivery**. Boston: Addison-Wesley Professional, 2017. 288 p.

APÊNDICE(S)

APÊNDICE A – Artigo

Integração de Ferramentas Open-Source para Elaboração de Infraestrutura para Implantação Contínua com Foco em Ambientes de Desenvolvimento Java

Kevin de Faveri Aguiar¹, Fabrício Giordani²

¹Acadêmico do Curso de Ciência da Computação – Unidade Acadêmica de Ciências, Engenharias e Tecnologias – Universidade do Extremo Sul Catarinense (UNESC) – Criciúma – SC

²Professor do Curso de Ciência da Computação – Unidade Acadêmica de Ciências, Engenharias e Tecnologias – Universidade do Extremo Sul Catarinense (UNESC) – Criciúma – SC

kevinfaveri@hotmail.com, fgiordani@gmail.com

Abstract. *Software development has several steps ranging from synchronization to a version control repository to delivery to the final user. The present work proposes the application of the agile practice of continuous implantation in a Java development environment for web to drastically eliminate manual processes and automate the entire process of software delivery. For definitions of the steps of a continuous deployment infrastructure, a bibliographical survey was carried out together with a search for open-source tools that meet the concepts that the agile practice demands. The work resulted in the automation of the entire process of delivering Java software to the web in the production environment.*

Resumo. *O desenvolvimento de software possui diversas etapas que vão desde a sincronização a um repositório de controle de versão até a entrega para o usuário final. O presente trabalho propõe a aplicação da prática ágil de implantação contínua em um ambiente de desenvolvimento Java para web visando eliminar drasticamente processos manuais e automatizar todo o processo de entrega de software. Para definição das etapas de uma infraestrutura de implantação contínua foi realizado um levantamento bibliográfico em conjunto a uma pesquisa de ferramentas open-source que atendem aos conceitos que a prática ágil exige. O trabalho trouxe como resultado a automatização de todo o processo de entrega de softwares Java para web em ambiente de produção.*

1. Introdução

A metodologia ágil é uma das práticas de desenvolvimento mais populares do momento, sendo utilizada por grandes empresas do ramo como Google e Microsoft. O desenvolvimento ágil se destaca por ajudar a entregar no software dentro do prazo e orçamento, com todas as funcionalidades originalmente prometidas ao cliente através da utilização de práticas como a integração contínua (SHORE, 2008, tradução nossa).

A integração contínua se destaca por ajudar a atenuar diversos problemas nas etapas de desenvolvimento, como a integração de uma parte do código ao resto do software ou casos onde o cliente presencia um problema que não é possível de ser reproduzido pela equipe, por exemplo, através da automatização de etapas de builds, testes e análises (KAWALEROWICZ; BERNTSON, 2011, tradução nossa).

Porém, mesmo apesar da grande difusão da metodologia ágil e suas práticas, como a integração contínua, as equipes de desenvolvimento ainda buscam conseguir um processo que seja previsível e de fácil reprodução, permitindo a implantação em produção de forma fácil e contínua (GAUDIN, 2015, tradução nossa).

Com isso, mediante o estudo da metodologia ágil de software e conseqüentemente a estrutura de uma *pipeline* de implantação contínua verifica-se a possibilidade modelar uma completa infraestrutura de desenvolvimento com o uso de ferramentas open-source.

2. Modelagem da Infraestrutura para Implantação Contínua

Este trabalho aqui proposto enfatiza a elaboração de uma infraestrutura para implantação contínua com foco em ambientes de desenvolvimento Java através da integração de ferramentas open-source utilizando como base a ferramenta de integração contínua Jenkins.

Tal infraestrutura, no âmbito da engenharia de software, proporciona a automatização de toda a *pipeline* de desenvolvimento, cobrindo desde o versionamento de código em um repositório até a entrega do software em ambiente de produção para o cliente final. De forma prática permite que os desenvolvedores se foquem inteiramente na escrita do código de novas funcionalidades que agreguem valor ao software o que, por consequência, acaba reduzindo o tempo necessário para a entrega de novas versões do software ao usuário final.

De forma a implementar o presente projeto foi estruturado um ambiente de desenvolvimento Java e utilizado o software *open-source* VirtualBox para virtualização de uma máquina virtual, esta última a qual serviu de hospedeira para as as ferramentas de forma que a infraestrutura para implantação contínua pudesse ser modelada.

A seguir foi escolhido o projeto em Java para web Spring Boot Petclinic, um projeto *open-source* da fundação Pivotal, em virtude de suas atualizações constantes e completude quando comparado a um projeto comercial grande ele se mostra o ideal para teste da infraestrutura a ser desenvolvida.

2.1 Ferramenta de Containerização

A ferramenta de containerização Docker foi escolhida como alternativa para execução da cada ferramenta da *pipeline* de implantação contínua pois ela isola cada ferramenta com relação as outras em um sistema operacional dedicado ao mesmo tempo em que possibilita a economia de recursos, pois todos os contêineres compartilham os recursos da máquina virtual hospedeira.

2.2 Servidor de Integração

Como solução para o servidor de integração foi utilizado o Jenkins, pois é uma alternativa *open-source* que possui um excelente suporte ao seu desenvolvimento, acumulando mais de dez mil estrelas de favorito em seu repositório principal (GITHUB, 2018), o que possibilita ao sistema ter uma natureza que facilite a integração a ferramentas externas. A instalação foi feita conforme o comando exposto na figura 1.

```
docker run --restart always -dit -p 50000:8080 -p 50001:50000 --name jenkins \
-e TZ=America/SaoPaulo --env JAVA_OPTS="-Xmx2048m -XX:MaxPermSize=512m" \
-v /opt/docker-volumes/jenkins:/var/jenkins_home \
-v /var/run/docker.sock:/var/run/docker.sock jenkins/jenkins:lts
```

Figura 1. Comando para execução do contêiner do Jenkins

Após a execução do contêiner e acesso a interface do Jenkins deve-se confirmar a instalação dos *plugins* recomendados e efetuar o cadastro de um usuário e senha que serão as credenciais utilizadas em futuros acessos a ferramenta. Após o fim da instalação deve-se voltar ao terminal da máquina virtual e executar os comandos da figura 2 na ordem apresentada afim de instalar o Docker dentro do Jenkins, o que possibilitará ao sistema criar novos contêineres por demanda.

```
1 - docker exec -it --user root jenkins /bin/bash

2 - apt-get update && \
apt-get -y install apt-transport-https \
ca-certificates \
curl \
gnupg2 \
software-properties-common && \
curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")/gpg > /tmp/dkey; apt-key add /tmp/dkey && \
add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") \
$(lsb_release -cs) \
stable" && \
apt-get update && \
apt-get -y install docker-ce

3 - usermod -a -G docker jenkins
```

Figura 2. Instalação do Docker dentro do contêiner do Jenkins.

2.3 Repositório de Código e Controle de Versão

De forma a executar o sistema de gerenciamento de repositório Gitlab (que já possui o Git integrado) na estrutura que vem sendo seguida por este trabalho deve-se utilizar o comando conforme figura 2 no terminal da máquina virtual, onde é declarado as portas pelas quais o sistema estará acessível, fuso horário e pastas onde serão persistidos os arquivos.

```
docker run -dit --publish 50100:80 --hostname 192.168.76.5:50100 \
  --publish 50101:443 --publish 50102:22 -e TZ=America/Sao_Paulo --name gitlab \
  --restart always --volume /opt/docker-volumes/gitlab/config:/etc/gitlab:Z \
  --volume /opt/docker-volumes/gitlab/logs:/var/log/gitlab:Z \
  --volume /opt/docker-volumes/gitlab/data:/var/opt/gitlab:Z gitlab/gitlab-ce:latest
```

Figura 3. Comando de execução do Gitlab.

Após a instalação o projeto que será usado para validar a infraestrutura de implantação contínua deverá ser sincronizado a um repositório de código local, facilitando uma posterior integração ao servidor de integração. Por fim, para integrar as funcionalidades do Gitlab ao servidor de integração é necessário instalar o *plugin* do mesmo pelo caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*, então selecionando *Gitlab*.

2.4 Ferramenta de *Build* de Projetos

Para que os projetos possam ser compilados em um executável é necessária uma ferramenta que execute o *build* dos projetos e nesta infraestrutura de implantação contínua foi escolhida dar suporte a utilização do Maven, que foi a ferramenta utilizada do projeto web. Para instalar a ferramenta no Jenkins deve-se acessar o mesmo pelo endereço de IP local configurado previamente, então na tela inicial do sistema deve-se clicar em *Gerenciar Jenkins* no menu lateral e então em *Global Tools Configuration* na lista que é apresentada. A seguir deve-se procurar na lista de ferramentas a palavra *Maven*, e configurar o nome e versão de instalação.

2.5 Ferramenta de Testes Unitários

A execução de testes unitários será feita com a utilização do Junit, que é a ferramenta de testes utilizada pelo projeto Java para web. Para que uma pipeline de implantação contínua possa executar testes unitários desta ferramenta será necessária a instalação de um *plugin* ao Jenkins, caso ainda não instalado, chamado *JUnit Plugin*, feita acessando o caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*.

2.6 Ferramenta de Testes Funcionais

Para a tarefa de execução de testes funcionais, onde componentes do sistema web são testados simulando o comportamento humano, será utilizada a ferramenta Selenium, que é a alternativa para testes deste tipo em sistemas Java para web. Aqui entra um empecilho com relação a utilização de contêineres: os mesmos não possuem telas, portanto o Jenkins não deverá conseguir executar os testes do Selenium. Como forma de contornar o problema será montada uma tela virtual, e para tal, deverá ser criada uma variável de ambiente no Jenkins. O usuário deverá fazer o caminho *Gerenciar Jenkins > Configurar o Sistema > Propriedades Globais* e clicar em *Variáveis de Ambiente*, adicionando uma nova variável com nome *DISPLAY* e valor *:0*. Por fim o usuário deverá executar no terminal da máquina virtual o comando demonstrado na figura 4.

```
docker exec jenkins Xvfb :0 >& /dev/null &
```

Figura 4. Comando para criar uma tela virtualizada no contêiner do Jenkins

A execução de testes do Selenium irá ocorrer em um navegador, e para que os mesmos executem com sucesso será necessário instalar o Google Chrome dentro do contêiner do Jenkins. Deverá ser então escrita a classe Java que será o teste de aceitação da *pipeline*, a qual

irá validar a possibilidade de clique no botão que possui o título *find owners* e caso não o encontre, irá diagnosticar um erro na aplicação. A classe em questão define o caminho em que o *driver* do Google Chrome (este definido no parágrafo anterior) e, caso o botão exista, mostra no *log* da *pipeline* para qual endereço o usuário seria redirecionado ao clicar. É importante salientar que será necessário adicionar no projeto que será utilizado para validar a infraestrutura de implantação contínua a biblioteca do Selenium.

Após a escrita da classe de testes será necessário baixar o *driver* do Google Chrome para Linux dentro do projeto em *resources/chromedriver*, esta que é uma ferramenta auxiliar que possibilita que comandos de clique sejam dados pelo Selenium dentro do navegador.

2.7 Ferramenta de Análise de Qualidade de Código

De forma a assegurar que convenções de código sejam seguidas à risca em grandes equipes de desenvolvimento existem ferramentas que analisam a qualidade de código, como o Checkstyle ou ferramentas mais completas, como o Sonarqube, que permite a geração de relatórios e melhor análise através da atribuição de notas para cada projeto. Além disso esta última ferramenta disponibiliza um *plugin* de integração ao Jenkins chamado *SonarQube Scanner* que deve ser instalado através do caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*.

Para executar a ferramenta em um container seguindo o que vem sendo condicionado como estrutura no decorrer deste trabalho deve-se executar o comando conforme figura 5 no terminal da máquina virtual, o qual define as pastas em quais serão persistidos os dados e as portas por quais a mesma estará disponível. Após o fim da execução do comando a ferramenta deverá ter sua interface disponível para acesso através da porta 50300 no IP da máquina virtual, podendo ser acessada de qualquer navegador na máquina local.

```
docker run -d --name sonarqube -p 50300:9000 -p 50301:9092
-v /opt/docker-volumes/sonarqube/data:/opt/sonarqube/data
-v /opt/docker-volumes/sonarqube/extensions:/opt/sonarqube/extensions sonarqube
```

Figura 5. Comando para execução da ferramenta SonarQube

Na interface do sistema o usuário deverá ser apresentado a uma tela de *login*, a qual permitirá o acesso com as credenciais *admin* como nome de usuário e *admin* como senha. A primeira tela da ferramenta deverá apresentar uma solicitação de nome para o *token* de autenticação que será gerado para integrações a outras ferramentas, não existindo um nome ideal pré-definido, ficando então a cargo do usuário a escolha do mesmo. Após esta etapa o sistema informará uma sequência de caracteres (*token* de autenticação) os quais devem ser copiados para serem utilizados na etapa de integração ao Jenkins.

Para conectar a ferramenta ao servidor de integração deve-se efetuar a instalação do *plugin SonarQube Scanner* conforme já informado, e então acessar na interface do Jenkins sua configuração através do caminho *Gerenciar Jenkins > Global Tool Configuration > seção SonarQube Scanner*. Deve-se então clicando em *Adicionar SonarQube Scanner* e definir o *Name*, este que será utilizado para referenciar o *plugin* durante a *pipeline*.

Por fim a integração do SonarQube precisa ser configurada no Jenkins, o que pode ser feito acessando o caminho *Gerenciar Jenkins > Configurar o Sistema > seção SonarQube servers*. Nesta tela deve-se marcar a caixa *Enable injection of SonarQube server configuration as build environment variables*, informar um nome (campo *name*) e o endereço em que se encontra a ferramenta (campo *Server Url*), e informar a sequência de caracteres gerada na etapa

anterior no campo *Server authentication token*. Com isso a ferramenta estará devidamente instalada e sua conexão com o servidor da integração configurada.

2.8 Ferramenta de Repositório de Artefatos

Ao final da *pipeline* de implantação contínua, logo antes da implantação em ambiente de produção, é necessário que o artefato gerado no processo seja arquivado para versionamento de versões implantadas com sucesso. Para resolver este problema existe a ferramenta Nexus, que é de utilização livre. Para execução da ferramenta deve-se utilizar o comando demonstrado na figura 6 no terminal da máquina virtual, o qual declara as pastas em quais os dados serão persistidos e a porta na qual sua interface será exposta.

```
docker run -d -p 50200:8081 --name nexus
-v /opt/docker-volumes/nexus:/sonatype-work sonatype/nexus
```

Figura 6. Comando para execução da ferramenta Nexus

Após a execução a ferramenta deverá ter sua interface disponível no IP local da máquina virtual na porta 50200 e poderá ser acessada usando *admin* como nome de usuário e *admin123* como senha.

Para efetuar a conexão da ferramenta ao servidor de integração deve-se instalar o *plugin Nexus Platform* através do caminho *Gerenciar Jenkins > Gerenciar Plugins > aba Disponíveis*. Após a instalação do *plugin* deve-se seguir o caminho *Gerenciar Jenkins > Configurar o Sistema* e então procurar pela seção *Sonatype Nexus*. Deve-se então informar valores a escolha do usuário para os campos *Nome de Exibição* e *DisplayID*. Para finalizar a configuração deve-se informar o endereço de IP da máquina virtual na porta exposta para a ferramenta Nexus em *Server URL* e as credenciais citadas no parágrafo acima nos respectivos campos. Ao confirmar a configuração o Jenkins estará devidamente integrado ao repositório de artefatos Nexus.

2.9 Contêiner Java para Execução do Projeto

Para que a execução do projeto web possa ocorrer será necessário criar uma imagem de contêiner que suporte a execução de aplicações Java. Para isto será necessário criar um arquivo de nome *Dockerfile* conforme a figura 7 dentro da máquina virtual que possui o Docker.

```
FROM openjdk:8
ENTRYPOINT ["sh", "-c", "java -jar /opt/spring-app/app.jar"]
RUN mkdir -p /opt/spring-app/
```

Figura 7. Arquivo Dockerfile

Por fim, para gerar a imagem de forma que ela possa ser executada futuramente para servir a aplicação web a ser utilizada nessa infraestrutura de implantação contínua deve-se chamar a funcionalidade de *build* do Docker no terminal da máquina virtual, definindo então o nome do mesmo como *java-app* (conforme figura 8).

```
docker build -t "java-app" .
```

Figura 8. Comando para build do contêiner “java-app”

2.10 Criação da *Pipeline* de Implantação Contínua

Com a instalação e configuração de todas as ferramentas necessárias para cada etapa da *pipeline* de implantação contínua é preciso, enfim, escrever o arquivo que descreve como a mesma será executada, o *Jenkinsfile*. O arquivo deverá ter a declaração de uma chave chamada *pipeline* a qual englobará toda a estrutura do arquivo e então definir o *agent* (que é o servidor de integração Jenkins no qual o procedimento será executado) como *any*, já que na infraestrutura atual temos somente um Jenkins rodando. Para finalizar a estrutura inicial do arquivo (figura 9) é preciso também definir que será utilizada a ferramenta Maven durante as etapas utilizando a chave *tools*, sendo o nome o mesmo que foi definido na configuração da ferramenta de *build* nas configurações do sistema do Jenkins.

```
pipeline {
  agent any
  tools {
    maven 'Maven'
  }
}
```

Figura 9. Estrutura inicial do Jenkinsfile

Para iniciar a declaração das etapas da *pipeline* de implantação contínua será necessário utilizar a chave *stages*, a qual conterá uma chave *stage* para cada etapa. Como primeira etapa a ser executada no processo foi definida a etapa de *build* automatizado e então implantação em um contêiner do projeto Java para web para que a etapa seguinte, que inclui testes funcionais, possa ser executada sem problemas, conforme figura 10.

```
stage('Build Automatizado e Implantacao em Container de Testes') {
  steps {
    echo '---> Iniciando compilacao do projeto em um container teste...'
    sh 'mvn package -DskipTests'
    sh 'docker create -p 60000:8090 --name java-test java-app'
    sh 'docker cp dist/app.jar java-test:/opt/spring-app/app.jar'
    sh 'docker network connect tcc-network java-test'
    sh 'docker start java-test'
    echo '---> Projeto para testes compilado e rodando com sucesso!!!'
  }
}
```

Figura 10. Estágio de build automatizado

A próxima etapa, como já citada, irá se tratar a definição da execução de testes unitários com JUnit e testes funcionais com Selenium. Para isso deverá ser adicionada mais uma chave *stage* conforme a figura 11.

```
stage('Testes (Unitarios e Funcionais/Aceitacao)') {
  steps {
    echo '---> Iniciando o testes do projeto...'
    sh 'mvn test'
    echo '---> Projeto testado com sucesso!!!'
    echo '---> Deletando container para execucao de testes...'
    sh 'docker rm -f java-test || true'
    echo '---> Container para testes eliminado com sucesso!!!'
  }
}
```

Figura 11. Estágio para execução de testes unitários e funcionais

Em caso de sucesso na execução dos testes, conforme a *pipeline* de implantação contínua já definida, deve-se ocorrer a análise de qualidade de código, a qual irá fazer uma avaliação do código escrito e salvar o resultado atribuindo uma nota, que ficará registrada no Jenkins. O *stage* que será então escrito deverá definir a utilização do *SonarQube Scanner* conforme figura 12.

```
stage('Análise de Qualidade de Código - SonarQube') {
    steps {
        echo '---> Iniciando análise de qualidade de código...'
        script {
            scannerHome = tool 'SonarQube';
        }
        withSonarQubeEnv('SonarQube') {
            sh 'mvn package -DskipTests sonar:sonar'
        }
        echo '---> Análise de qualidade de código encerrada...'
    }
}
```

Figura 12. Estágio para análise da qualidade de código

Após a análise de qualidade de código e registro de seus resultados deve-se utilizar o repositório de artefatos Nexus para guardar o executável gerado pela *pipeline*, sendo então necessário adicionar um *stage* com o comando que chama o *plugin* do Nexus que salva o artefato conforme representado na figura 13.

```
stage('Repositorio de Artefatos - Nexus') {
    steps {
        echo '---> Iniciando o Upload do artefato gerado...'
        nexusPublisher nexusInstanceId: 'localNexus',
            nexusRepositoryId: 'releases', packages: [[class: 'MavenPackage',
            mavenAssetList: [[classifier: '', extension: '', filePath: 'dist/app.jar']],
            mavenCoordinate: [artifactId: 'spring-petclinic', groupId: 'org.springframework.samples',
            packaging: 'jar', version: '${BUILD_ID}']]
        echo '---> Upload do artefato executado e salvo com sucesso no repositório Nexus...'
    }
}
```

Figura 13. Estágio para salvar executável no repositório de artefatos Nexus

Por fim, após salvar o executável no repositório de artefatos, existe a etapa de implantação em ambiente de produção visando a entrega da aplicação Java para web para utilização por parte dos usuários. O novo *stage* (figura 14) a ser adicionado deve efetuar a criação de um contêiner Docker que utilize a imagem *java-app* criada anteriormente, então copiando o executável Java gerado por esta *pipeline* e colocando dentro do contêiner.

```
stage('Implantacao em Ambiente de Produção') {
    steps {
        echo '---> Iniciando implantação em ambiente de produção...'
        sh 'docker create -p 8080:8090 --name java-app java-app'
        sh 'docker cp dist/app.jar java-app:/opt/spring-app/app.jar'
        sh 'docker start java-app'
        echo '---> Projeto implantado em ambiente de produção com sucesso!!!'
        echo '---> Está disponível na porta 8080 do host do Docker (http://192.168.76.5:8080)!!!'
    }
}
```

Figura 14. Estágio para implantação em ambiente de produção

Com a escrita dessas adições ao arquivo *Jenkinsfile* (que deverá ter efetuado seu *commit* para o repositório de código) toda a *pipeline* de implantação contínua proposta estará descrita em etapas (que são os *stages* definidos no arquivo) e o Jenkins estará apto a executar a mesma.

3 Resultados Obtidos

Com a integração das ferramentas em um servidor de integração foi possível estruturar uma infraestrutura de implantação contínua com uma *pipeline* objetivando a entrega um projeto Java para web em ambiente de produção sem a interferência do programador e validar cenários de sucesso e quebra.

Um cenário de sucesso ocorre quando todos os testes não identificam quaisquer erro e o *build* automatizado e entrega do software em ambiente de produção é feito com sucesso. Nessa infraestrutura foi possível validar esse cenário com o projeto Java para web escolhido, conforme figura 15.

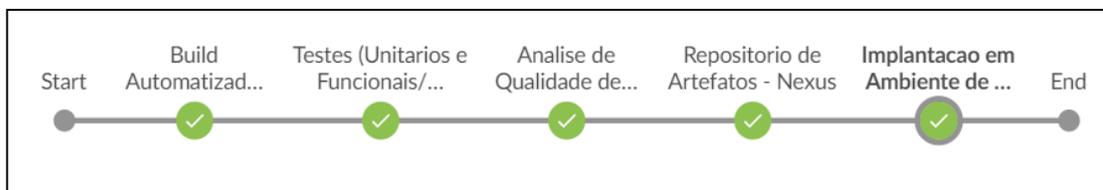


Figura 15. Cenário de Sucesso na *Pipeline* de Implantação Contínua

Um cenário de quebra ocorre quando algum teste identifica algum erro ou o *build* e implantação não pode ser realizado por motivos diversos, como falha na máquina. Nessa infraestrutura foi possível validar esse cenário com o projeto Java para web escolhido, conforme figura 16.

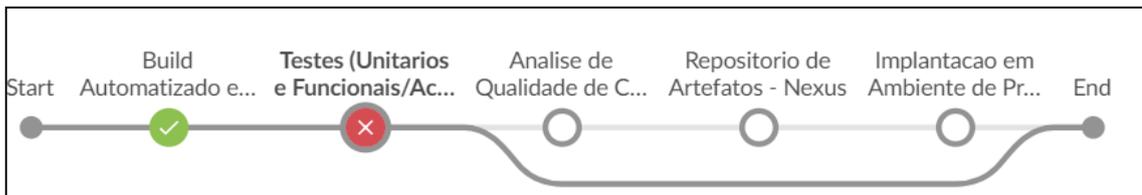


Figura 16. Cenário de Quebra na *Pipeline* de Implantação Contínua

4 Conclusão

Com o emprego do Jenkins (e as ferramentas integradas) na criação e execução de *pipelines* de implantação contínua foi possível automatizar todo o processo de entrega de software desde o *build* automatizado até que o mesmo chegasse ao ambiente de produção sendo então entregue ao usuário final.

É importante ressaltar também o aspecto de entrega contínua que a infraestrutura possibilitou, isto é, após uma pipeline estar configurada todos os *builds* posteriores irão apenas repetir um fluxo predefinido, o que transformou a dificuldade de se entregar o software devido a processos manuais e propensos a falha humana em algo inexistente.

Ao final foi possível validar o funcionamento de toda a infraestrutura, podendo serem observados cenários de sucesso, onde o software foi entregue em ambiente de produção e cenários de falha, onde o mesmo não passou em alguma classe de teste escrita para cobrir erros no software, então documentando a causa do erro para consulta por parte dos desenvolvedores.

Uma observação importante diz respeito ao fato de que foi designado um foco em configurar uma infraestrutura que fosse agnóstica com relação a *frameworks* utilizados no desenvolvimento do projeto Java para web visando aumentar os cenários em que a mesma pudesse ser utilizada. Dada a forma com a qual a infraestrutura foi modelada é possível afirmar que a mesma pode atender a qualquer software desenvolvido em Java para web que utilize as mesmas ferramentas de *build* e testes deste trabalho, e em caso contrário, sendo necessárias somente pequenas alterações, como por exemplo a instalação de um *plugin* para a ferramenta divergente ao Jenkins.

Com isto, é possível afirmar que todos os objetivos foram alcançados, o que conseqüentemente proporcionou a elaboração de uma infraestrutura de implantação contínua plenamente funcional.

References

GAUDIN, Olivier. Prefácio. In: DAVIS, Christopher W. H.. **Agile Metrics in Action**. Greenwich, Connecticut: Manning Publications, 2015. 272 p.

GITHUB. Jenkins automation server . Disponível em: <<https://github.com/jenkinsci/jenkins>>. Acesso em: 01 de mai. 2018.

KAWALEROWICZ, Marcin; BERNTSON, Craig. **Continuous Integration in .NET**. Greenwich, Connecticut: Manning Publications, 2011. 375 p.

SHORE, James; WARDEN, Shane. **The Art of Agile Development**. Sebastopol, Ca: O'reilly Media, 2007. 432 p.