CENTRO UNIVERSITÁRIO DE ANÁPOLIS - UNIEVANGÉLICA
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

HEFRAYN ANTERO BARROS BARBOSA

ANÁLISE DAS VANTAGENS DO USO DE CI/CD NO DESENVOLVIMENTO DE UM PROJETO JAVA WEB

ANÁPOLIS
2018-2

HEFRAYN ANTERO BARROS BARBOSA

ANÁLISE DAS VANTAGENS DO USO DE CI/CD NO DESENVOLVIMENTO DE UM PROJETO JAVA WEB

Trabalho de Conclusão de Curso II apresentado ao Curso de Engenharia de Computação do Centro Universitário de Anápolis, como requisito para obtenção do grau de Bacharel em Engenharia de Computação.
Orientador Prof. Me.: Marcelo de Castro Cardoso

ANÁPOLIS

2018-2

HEFRAYN ANTERO BARROS BARBOSA

ANÁLISE DAS VANTAGENS DO USO DE CI/CD NO DESENVOLVIMENTO DE UM PROJETO JAVA WEB

Trabalho de Conclusão de Curso II apresentado como requisito parcial para a obtenção de grau do curso de Bacharelado em Engenharia de Computação do Centro Universitário de Anápolis – UniEVANGÉLICA.

Aprovado(a) pela banca examinadora em 05 de Dezembro de 2018, composta por:

Marcelo de Castro Cardoso
Presidente da Banca

Natasha Sophie Pereira
Prof(a). Convidado(a)

Walquíria Fernandes Marins
Prof(a). Convidado(a)

RESUMO

A evolução constante dos negócios da era digital está, a cada dia, requerendo novos métodos de desenvolvimento e entrega de *software*, o mercado vem reagindo e consolidando neste meio conceitos importantes para o bom relacionamento entre desenvolvedores e clientes. Este trabalho tem como objetivo realizar um estudo teórico sobre os conceitos de integração e implantação contínua de software, levantar quais as principais ferramentas disponíveis, analisar e escolher algumas ferramentas para a implementação dos conceitos e aplica-las no desenvolvimento um software JavaWeb, registrar as vantagens e desvantagens da sua utilização em projetos de pequeno porte.

Palavras-Chave: Integração contínua; Implantação contínua; Controle de versão; Kubernets

ABSTRACT

The constant evolution of the digital age business is, every day, requiring new methods of software development and delivery, the market has been reacting and consolidating in this medium important concepts for the good relationship between developers and customers. This work aims to carry out a theoretical study on the concepts of integration and contínuous deployment of software, raising the main tools available, analyzing and choosing some tools for the implementation of concepts and Apply them in the development of Javaweb software, registering the advantages and disadvantages of your using in small projects.

Keywords: Contínuous Integration; Contínuous Deployment; Version Control; Kubernets

LISTA DE SIGLAS

API: Application Programming Interface.

CD: Contínuous Deploy.

CI: Contínuous Integration.

CLI: Command Line Interface.

CPU: Central Processing Unit.

CRUD: Create, Read, Update, Delete.

DPL: Ferramenta de Deploy feita para contínuous deploy.

HTML: Hypertext Markup Language.

IBM: International Business Machines.

IDE: Integrated Development Environment.

JAR: Java ARchive.

PaaS: Plataform as a Service.

PHP: Personal Home Page - Hypertext Preprocessor.

POM: Project Object Model.

REST: Representational State Transfer.

SGBD: Sistema Gerenciador de Banco de Dados.

SO: Sistema Operacional.

SQL: Structured Query Language.

MB: Mega Byte.

MVC: Model View Controller

MySQL: é um sistema gerenciador de banco de dados relacional

NoSQL: Banco de dados não relacional.

TB: Tera Byte

TDD: Test Driven Development.

VM: Virtual Machine.

XML: Extensible Markup Language.

XP: Extreme Programming.

YAML: YAML Ain't Markup Language.

LISTA DE FIGURAS

Figura 1 - Ciclo TDD 20

Figura 2 - Contínuous Delivery / Contínuous Deployment 25

Figura 3 - DB-Engines Ranking 34

Figura 4 - DB-Engines Tendência de popularidade 35

Figura 5 - Analise Comparativa Repositório de Código 39

Figura 6 - Heroku vs. Google App Engine vs. AWS 40

Figura 7 - Orçamento Cirúrgico 42

Figura 8 - Diagrama de Caso de Uso 43

Figura 9 - Diagrama de Classes 43

Figura 10 - Arvore de Branchs 46

Figura 11 - .gitlab-ci.yaml 47

Figura 12 - Testes Unitários CRUD Paciente 48

Figura 13 - Pipeline Failed 48

Figura 14 - Pipeline Passed 49

Figura 15 - Deploy Heroku 1 50

Figura 16 - Deploy Heroku 2 50

Figura 17 - NewBudget Implantado Heroku 51

Figura 18 - Pipeline Deploy Automatizado 52

Figura 19 - Merge Médico/Paciente Deploy Passed 52

Figura 20 - NewBudget Cadastrar Médicos 53

Figura 21 - Kucetl Get Nodes 54

Figura 22 - Dockerfile 55

Figura 23 - Kubernets Implantado com Falha 55

Figura 24 - Kubernets Implantado com Sucesso 56

Figura 25 - NewBudget Implantado Kubernets 57

Figura 26 - Estatísticas Commits 59

SUMÁRIO

1. INTRODUÇÃO 12
1.1. PROBLEMA 12
1.2. OBJETIVO GERAL 13
1.2.1. Objetivos Específicos 13
1.3. JUSTIFICATIVA 13
1.4. METODOLOGIA 14

2. FUNDAMENTAÇÃO TEÓRICA 16
2.1. METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE 16
2.1.1. XP - Extreme Programming 17
2.1.2. TDD - Test Driven Development 19
2.2. TESTES DE SOFTWARE 21
2.2.1. Testes Unitários 22
2.3. CI / CD 23
2.3.1. Integração Contínua - CI 23
2.3.2. *Deploy* Contínuo - CD 25
2.3.3. Controle de Versão 26

3. MÉTODOS E FERRAMENTAS UTILIZADAS 28
3.1. FERRAMENTAS PARA CI/CD 28
3.1.1. Repositório de Código 28
3.1.2. Docker 29
3.1.3. Orquestradores de Contêineres 30
3.1.4. Ferramentas de Build 31
3.1.5. Ambientes de Deploy 31
3.2. AMBIENTE DE DESENVOLVIMENTO 32
3.2.1. IDE Java 32
3.2.2. Framework Spring Boot 33
3.2.3. Sistema Gerenciador de Banco de Dados 34
3.2.4. Framework Front End 36

3.2.5. Metodologia de Desenvolvimento 37
3.2.6 Aplicação do Framework XUnit 38
3.3. INTEGRAÇÃO CONTÍNUA 38
3.4. DEPLOY CONTÍNUO 39

4. PROJETO DE SOFTWARE 42
4.1. DESCRIÇÃO DO SISTEMA 42
4.2. DIAGRAMA DE CASO DE USO / DIAGRAMA DE CLASSES 43
4.3. A IMPLEMENTAÇÃO DO SOFTWARE 44
4.3.1. Uso do GitLab CI 45
4.3.2. Exploração dos Testes de Softwares 47
4.3.3. Entrega Contínua - Deploy Manual 49
4.3.4. Implantação Contínua - Deploy automatizado 51
4.3.5. Implantação CD com Kubernets 53

5. CONSIDERAÇÕES FINAIS 58

7. REFERÊNCIAS BIBLIOGRÁFICAS 61

1. INTRODUÇÃO

O *software* está dia após dia se mostrando presente nas mais variadas atividades realizadas pelo ser humano. Tarefas simples, como compras em supermercados, olhar as horas no relógio, checar os parâmetros do carro no painel, dentre outras aplicações mais sofisticadas como olhar o saldo em conta bancária. Porém, mesmo com essa diversidade de aplicações do *software*, é possível que a maioria das pessoas já tenha passado pela experiência de utilizar *softwares* que não se comportaram da maneira para a qual foram projetados. Esses problemas podem se transformar em problemas maiores, como prejuízos financeiros podendo, em alguns casos, denegrir a reputação de pessoas ou empresas (MULLER et al apud LIMA, 2011).

1.1. PROBLEMA

Segundo Muller et al (apud LIMA, 2011), as pessoas podem cometer erros, o erro humano gera um defeito, que no código de um sistema em desenvolvimento, ao ser executado, pode gerar uma falha resultando em um comportamento inesperado do *software*.

Para reduzir problemas dessa magnitude é importante compreender a causa dos defeitos e, para isso, rastreamento e controle são necessários do início do projeto até o término de operação do *software*. Com foco na qualidade do *software* empresas buscam aperfeiçoar processos de desenvolvimento de *software* através da inclusão de testes, pois, a possibilidade de mudança nos requisitos não pode ser descartada (PRESSMAN, 2011).

Segundo Pressman (2011) atividades de gerência de configuração e testes dão suporte ao controle de alterações dos códigos, além de assegurar que o *software* irá se comportar da maneira correta, levando para o caminho da integração contínua.

Um projeto que implemente os conceitos da integração contínua, assume que o *software* deve ser testado continuamente de forma automática. Caso mudanças ocasionem erros, estes devem ser identificados imediatamente após a sua criação

para serem corrigidos, sendo possível integrar o código em desenvolvimento ao ambiente de testes/homologação várias vezes ao dia (FARLEY; HUMBLE, 2014).

Startups no mundo inteiro iniciam suas atividades com projetos pequenos que podem evoluir rendendo um bom produto. Diante deste cenário é possível se perguntar quais as vantagens e desvantagens do uso da integração e *deploy* contínuos em projetos de pequeno porte?

1.2. OBJETIVO GERAL

Realizar um estudo bibliográfico para identificar os requisitos de uma boa implementação da integração contínua e *deploy* contínuo (CI/CD) aplicado ao desenvolvimento de um sistema Java Web, documentando os principais benefícios.

1.2.1. Objetivos Específicos

- Mapear as principais ferramentas existentes no mercado para CI/CD
- Analisar e implementar ambientes de CI/CD e testes automatizados
- Apresentar os resultados com uma visão crítica dentro do processo de desenvolvimento.

1.3. JUSTIFICATIVA

Empresas de *software* estão sempre procurando novas formas de aumentar suas capacidades de produção através da medição e controle do desempenho de suas equipes utilizando métodos ágeis (SOMMERVILE, 2003).

Para tanto é necessária a melhoria nos processos de desenvolvimento de software em pontos específicos através de diversas ferramentas, tais como, testes, documentação, metodologias ágeis etc. Muitas vezes *frameworks*[1] melhoram o desempenho do processo de desenvolvimento em pontos específicos para o qual foram desenvolvidos, deixando gargalos em outros pontos não abordados pelo *framework*, acarretando em ganhos pequenos de produtividade devido à falta de integração das ferramentas utilizadas.

[1] Modelos *templates* com diversas funções que podem ser usadas pelo desenvolvedor para economizar tempo de desenvolvimento.

Em linhas gerais os conceitos de CI/CD visam a automação do processo de desenvolvimento de *software* do início do desenvolvimento até a entrega do software em produção, inclusive durante a manutenção de *software*, garantindo versatilidade de mudanças e entregas rápidas de novas funcionalidades (FOWLER, 2006).

Sabemos que a aplicação de alguns conceitos no ciclo de desenvolvimento e evolução do *software* visam diminuir o risco de perda de qualidade, contudo não é possível garantir isso (ARAÚJO, TRAVASSOS, 2006).

Este trabalho tem o intuito é demonstrar o quão eficaz pode se tornar a aplicação dos conceitos de CI/CD em projetos de pequeno porte, visto que a configuração de ambientes leva um tempo considerável.

1.4. METODOLOGIA

Para alcançar os objetivos propostos, foi realizado um estudo para identificar as melhores práticas e tecnologias usadas na implementação dos conceitos de CI/CD, bem como a aplicação dessas tecnologias e práticas implementando os conceitos através da construção de um ambiente de desenvolvimento, testes e produção/homologação. Neste processo foram extraídos dados dos ambientes implantados para uma análise das vantagens e desvantagens ao se utilizar essas tecnologias em projetos de pequeno porte.

Os ambientes construídos utilizando CI/CD, foram aplicados a um projeto JavaWeb que já estava em desenvolvimento sem o uso das tecnologias. A aplicação do projeto nos ambientes deu subsidio para a extração de dados referentes ao desempenho de desenvolvimento, sendo possível analisar os prós e contras da utilização da CI/CD.

O próximo capitulo apresenta a teoria fundamental de CI/CD, onde foram discutidas metodologias de desenvolvimento, testes e o próprio CI/CD. O capitulo 3 apresenta uma busca pelas melhores ferramentas disponíveis para a implementação dos ambientes. No Capitulo 4, foi descrita as ferramentas utilizadas no ambiente de desenvolvimento, como foram aplicados os conceitos CI na ferramenta GitLab, a Implementação do CD no Heroku e no Google App Engine, sendo 1 ambiente de CI

no GitLab, 2 ambientes de CD no Heroku, e um ambiente de CD no Google App Engine. No capitulo 5 será tratado sobre as considerações finais.

2. FUNDAMENTAÇÃO TEÓRICA

Neste Capitulo serão abordados conceitos importantes para o tema deste trabalho, como: metodologias de desenvolvimento alinhadas aos conceitos de CI/CD, testes de *software*, bem como conceitos diretamente ligados ao CI/CD.

2.1. Metodologias de desenvolvimento de software

No início da primeira guerra mundial o mundo passava por grandes transformações, sendo que uma delas foi a necessidade de controlar os processos de trabalho fomentados pelo aumento do volume de dados das organizações de todos os setores. No final da segunda guerra, com o surgimento e dinamização dos computadores veio a necessidade de construção de *softwares* que automatizassem processos complexos das organizações. Neste cenário surgiram alguns modelos de desenvolvimento de *software* para simplificar o processo de desenvolvimento (UCHÔA, 2008)

Segundo Almeida (2017), as metodologias de desenvolvimento de software podem ser classificadas em metodologias tradicionais e metodologias ágeis, tais como descritas abaixo:

- Metodologias tradicionais:

> Metodologias que apresentam, entre outras características, fases bem definidas de ciclo de vida de desenvolvimento de software, maior documentação sobre o produto gerado e suas fases de desenvolvimento, maior formalização da comunicação através de documentação, papéis bem definidos para os membros da equipe e busca de padronização das etapas de desenvolvimento (ALMEIDA, 2017, p 11).

- Metodologias ágeis

> Metodologias que apresentam, entre outras características, etapas iterativas (vários ciclos incrementais) de desenvolvimento de software, menor documentação e maior foco no desenvolvimento do software ou sistema, comunicação mais informal entre os membros da equipe, papéis menos definidos ou rotação de papéis entre os membros e maior aceitação a mudanças que possam ocorrer durante o processo de desenvolvimento (ALMEIDA, 2017, p 11).

2.1.1. XP - Extreme Programming

Extreme Programming (XP) é uma metodologia ágil desenvolvida para equipes de desenvolvimento de pequeno e médio porte, criada para projetos que tenham requisitos vagos e com grandes possibilidades de mudanças. Os principais diferenciais do XP em relação as demais metodologias se resumem em *feedback* constantes, abordagem incremental e comunicação (BECK, 2004).

Segundo Teles (2004), o XP tem o objetivo de entregar ao cliente o máximo de resultados da equipe de desenvolvimento para cada dia trabalhado, onde a metodologia é organizada de forma a garantir que o cliente tenha um grande retorno do investimento no software.

Beck (2004) resume o XP em quatro atividades principais:

- **Codificar**: Baseado no principio de que se o desenvolvedor não codificar o projeto não anda.
- **Testar**: Se não houver testes não será possível saber quando parar de codificar. Se passou no teste é porque está funcionando.
- **Ouvir**: É necessário ouvir para o que codificar e como planejar os testes para atingir os objetivos da funcionalidade em desenvolvimento.
- **Projetar**: Para tornar os passos de Codificar, testar e ouvir em um ciclo constante de trabalho da equipe, focando sempre em projetos simples.

Os principais valores da metodologia são:

- **Comunicação**: para a implementação do feedback é imprescindível a comunicação constante e intensa entre a equipe, sendo que essa comunicação deve ser face a face, pois, esse tipo de comunicação é o mais eficaz na troca de informações. É dispensável comunicações por telefone ou mensagens instantâneas por serem, na maioria das vezes, ineficazes na troca de informações (TELES, 2004).
- **Simplicidade**: Projetos de software que os requisitos são altamente mutáveis requerem simplicidade no código para facilitar a manutenção e a velocidade de entrega de cada funcionalidade, sendo que o código deve estar disponível para uso logo após a codificação e testes. (BECK, 1999).

- **Respeito**: Esse é o valor que serve de base aos outros valores. Sem o respeito se torna inviável o uso da metodologia, que requer dos envolvidos no projeto entendimento dos diferentes pontos de vista.

As principais práticas são:

- **Cliente Presente**: No XP a interação do cliente com a equipe de desenvolvimento é muito importante para se obter o máximo de desempenho através dos feedbacks, sanando rapidamente as duvidas dos desenvolvedores alinhando rapidamente desvios na implantação das funcionalidades (TELES, 2004).
- **Planejamento**: O planejamento tem o objetivo de alinhar o desenvolvimento na direção das funcionalidades que são mais importantes para o cliente naquele momento, descrevendo resumidamente cada funcionalidade, deixando-o ciente do prazo que será gasto na implementação (TELES, 2004).
- **Pequenas Versões**: Garante que sejam realizados diversas entregas com novas funcionalidades em curtos períodos de tempo, criando um ambiente colaborativo e aumentando a relação de confiança entre a equipe de desenvolvimento e o cliente (BECK, 2004).
- **Metáfora**: Em um ambiente multidisciplinar como um ambiente de desenvolvimento de *software*, muitas vezes o cliente entende muito das regras de negócio e desenvolvedores entendem muito de programação e desenvolvimento podendo a comunicação ficar prejudicada, sendo necessário a utilização de metáforas para simplificar as ideias e informações transmitidas, facilitando o entendimento de todos (BECK, 2004).
- **Projeto Simples**: Quanto mais simples for o projeto do sistema mais fácil será adaptá-lo às mudanças, sendo que somente projetos simples são essenciais na redução de tempo e custos da mudança (BASSI, 2008).

- **Ritmo Sustentável**: É necessário respeitar os limites físicos dos envolvidos no projeto. Pessoas trabalhando cansadas não conseguem aplicar bem os conceitos do XP (GONÇALVES JR, 2009).
- **Posse Coletiva**: Acesso irrestrito de todos os programadores envolvidos ao código fonte, podendo realizar alterações no que for necessário para resolver eventuais problemas de forma célere (TELES, 2004).
- **Programação em pares**: Disponibilização de apenas um computador para um par (dois) de programadores. Esta prática traz vários benefícios como compartilhamento de conhecimento, revisão constante do código e redução de erros (TELES, 2004).
- **Padrões de Codificação**: a equipe de desenvolvimento deve adotar uma padronização do código desenvolvido conhecido por todos, facilitando assim as manutenções que serão rápidas e seguras (TELES, 2004).
- **Desenvolvimento Orientado a Testes**: É necessário a criação de um teste para cada funcionalidade, obrigando o programador a planejar as interfaces, classes e métodos, além de criar uma bateria de testes que poderá ser utilizado para validar todo o sistema em qualquer fase do projeto (TELES, 2004).
- **Refatoração**: Ao terminar uma funcionalidade, é importante o desenvolvedor revisar o código para garantir sua legibilidade para futuras alterações e/ou manutenções (GONÇALVES JR, 2009).
- **Integração Contínua**: Após a criação da funcionalidade o desenvolvedor deve integrá-la ao projeto principal, podendo ser feito quantas vezes for necessário ao longo do dia (BECK, 2004).

2.1.2. TDD - Test Driven Development

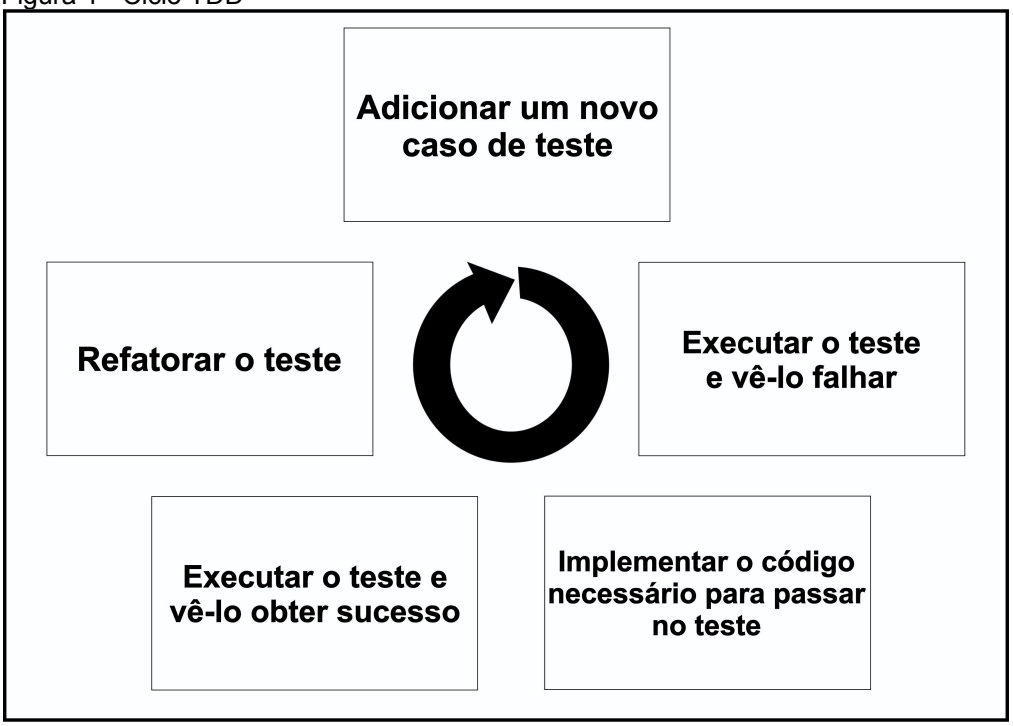
O TDD (*Test Driven Development* - Desenvolvimento Orientado a Testes) é uma metodologia ágil derivada do manifesto ágil e da metodologia XP (*Extreme Programming*) (BECK, 2000).

A metodologia é usada na etapa de desenvolvimento do *software*, onde os programadores usam testes para guiar o desenvolvimento. Na prática o desenvolvimento começa a partir da elaboração dos casos de testes de um objeto (Borges, 2006).

O TDD funciona com ciclos curtos, semelhante ao XP, onde o desenvolvedor elabora um teste para a função que deseja implementar ou alterar, realiza a execução do teste que irá falhar devido a falta de implementação da funcionalidade, realiza a implementação da funcionalidade da forma mais simples que conseguir focado em conseguir passar no teste, executa o teste e o mesmo deverá passar, refatorar o código deixando-o legível para prováveis manutenções futuras. Podemos ver o ciclo TDD na figura 1 (FEITOSA, 2007).

Fonte: adaptado

Figura 1 - Ciclo TDD



(BORGES,2006)

Segundo Gama (2010) uma das principais vantagens do TDD é que o programador tem a possibilidade de evoluir o código a passos pequenos e controlados, dessa forma caso ocorram erros será fácil localizar e corrigir o problema.

Segundo Beck (2000), alguns padrões devem ser seguidos na utilização da metodologia TDD, conforme descritos abaixo:

- **Testes Isolados**: Os testes devem ser livres de qualquer dependência, podendo ser executados a qualquer momento de forma rápida sem que este interfira no próximo teste.
- **Lista de Testes**: Antes de implementar qualquer coisa o programador deve ter uma visão do que será produzido ao final do trabalho, sendo necessário criar uma lista com todos os testes a serem desenvolvidos, esta lista servirá como guia de desenvolvimento.
- **Teste Primeiro**: A medida que um programador codifica ao longo do tempo o estresse aumenta gradativamente. Por esse motivo os testes devem ser escritos antes da codificação real da funcionalidade, evitando assim que os testes deixem de existir.
- **Dados de Teste**: Os dados de entradas nos testes devem ser escolhidos de forma a tornar os testes fáceis de ler, em caso de muitas entradas devem ser escolhidos dados que cubram cada categoria de entrada, tornando desnecessário, por exemplo, ter uma lista de 8 entradas se 4 cobrem todas as categorias.
- **Dados Evidentes**: Os testes devem ser construídos de forma que facilite a compreensão do que ele está testando e qual é o resultado esperado quando for executado.

2.2. TESTES DE SOFTWARE

O desenvolvimento de *software* é um processo de alta complexidade onde podem ocorrer muitos erros. Para minimizar os erros do processo de desenvolvimento, é importante adicionar ao processo etapas de verificação e validação de *software*, onde o principal recurso para verificar e validar o *software* é o teste (BARBOSA et al., 2000)

Segundo Pádua (2003), o teste de *software* se tornou parte integrante do processo de desenvolvimento de *software* de equipes que visam a qualidade de suas funcionalidades, sendo este, o recurso que provê métricas para medir a qualidade de um *software*.

Para Pedrycz e Peters (2001) o teste tem o dever de encontrar e reduzir erros, focando no ganho de qualidade ao software em desenvolvimento, além de ser a principal fonte de *feedback,* facilitando assim, a comunicação entre os envolvidos no projeto. Pressman (2011), diz que um bom planejamento de testes, resulta no sucesso da construção de um *software*.

Segundo Rocha et al. (2001) os testes podem ser divididos em etapas, tais como descritas abaixo:

- **Teste de unidade:** Popularmente conhecido como teste unitário, tem o objetivo de testar a menor parte lógica do *software*, focando os testes nos métodos de regras de negócio, em busca de erros lógicos
- **Teste de Integração:** Foca os testes nas interfaces de comunicação entre os módulos de um sistema quando saem do ambiente de desenvolvimento e se integram ao sistema.
- **Teste de Sistema:** Com uma visão de usuário, o teste de sistema busca verificar se foram satisfeitos os requisitos através da utilização do próprio sistema, simulando o mesmo ambiente do dia a dia do usuário final.
- **Teste de Aceitação:** Semelhante ao Teste de Sistema, o teste de aceitação é realizado por um restrito grupo de usuários, que verifica se as rotinas do sistema estão condizentes com os requisitos solicitados.
- **Teste de Regressão:** Não sendo considerado um nível de teste, o teste de Regressão pode ser considerado como uma estratégia para reduzir efeitos indesejados ao inserir no sistema em desenvolvimento novas funcionalidades e versões. Seu funcionamento se dá através da aplicação de todos os testes já aplicados em etapas anteriores do ciclo de vida.

2.2.1. Testes Unitários

Dentre as atividades de Integração contínua mais importantes estão os testes automatizados, que tem o objetivo de detectar rapidamente os erros de integração. Neste cenário é importante a equipe de desenvolvimento aderir à metodologia de

desenvolvimento boas práticas para evitar a negligência no desenvolvimento de testes (ROCHA, 2013).

Uma das técnicas mais utilizadas é a utilização dos *frameworks* de testes XUnits associado a metodologia de Desenvolvimento Orientado a Testes (TDD). O XUnits dispõe de varias variantes para diversas linguagens de programação como o JUnit para Java, o PHPUnit para PHP, CppUnit para C++, dentre outras variantes. O TDD é uma metodologia de desenvolvimento de ciclos curtos onde primeiro é desenvolvido o teste para depois desenvolver a funcionalidade (ROCHA, 2013).

O framework XUnit é um *framework* de testes unitários e cada teste implementado deve testar apenas uma funcionalidade de forma isolada sem depender de outros testes para seu correto funcionamento. Para identificar os métodos de testes o *framework* utiliza anotações no código e para validar os testes são utilizados asserções que são responsáveis por indicar ou não a falha do teste (ROCHA, 2012).

2.3. CI / CD

Os métodos tradicionais de desenvolvimento e entrega de software estão a cada dia entrando em desuso devido ao intervalo de tempo para essas atividades ocorrerem serem extensos. Com a demanda crescente por resultados esse tempo vem diminuindo para anos, semestres, trimestres, meses etc. Com o advento das tecnologias de Integração Contínua e Entrega Contínua essas entregas passam a ocorrer semanalmente, diariamente e até varias entregas em um mesmo dia (AZERI, 2017). Em seguida são apresentados os principais conceitos de CI/CD.

2.3.1. Integração Contínua - CI

O conceito de Integração Contínua teve origem na metodologia ágil XP em uma de suas doze práticas (FOWLER, 2006). A metodologia prega que para o projeto em desenvolvimento se adaptar bem a cada mudança no código, é essencial que seja feita a integração do código em ciclos menores e no menor tempo possível, para obter melhores ganhos em previsibilidade e redução das incertezas no projeto (MEDEIROS, 2013).

Segundo Fowler (2006) o conceito de integração contínua não requer nenhum tipo de ferramenta para ser implementado, podendo ser implementado manualmente, porém é importante agregar ao processo de integração contínua ferramentas para automatização do processo.

A implementação da integração contínua tem como pré requisito a utilização de um sistema de controle de versão, que tenha um repositório onde o código desenvolvido pelos programadores esteja concentrado, e que o código tenha um alto grau de testes unitários automatizados além de uma ferramenta que monitore constantemente o código para executar a construção e rodar os testes unitários (FOWLER, 2006).

Inicialmente o fluxo seguido pelo conceito de integração contínua, consiste na obtenção de uma cópia do código fonte do repositório central (*MainLine* ou *Branch Master*) por parte do programador para seu repositório local. Após realizar as alterações devidas no código é necessário que o programador realize o *build* do projeto localmente para a execução dos testes automatizados, que em caso de falha deve ser revisto imediatamente, voltando para a etapa de codificação. Somente quando todos os testes estiverem executados sem apresentar falhas ou erros que será possível integrar as alterações ao repositório central, sendo que será necessário que a ferramenta de integração contínua, quando for o caso, realize o mesmo processo de *build* e testes no repositório central para garantir que as funções inseridas não impactem funções inseridas por um segundo programador em caso de programação paralela (FOWLER, 2006).

Para Fowler (2006), em um ambiente com integração contínua, uma falha de construção não pode permanecer por muito tempo sem ser corrigida, sendo importante realizar diversas integrações corretas por dia, e ocasionais integrações com falhas devem ser rapidamente corrigidas. O resultado é um *software* funcional com baixo índice de *bugs*, com todos os programadores ancorados no código central estável, nunca se distanciando demais do repositório central para não dificultar a integração de volta (FOWLER, 2006).

2.3.2. *Deploy* Contínuo - CD

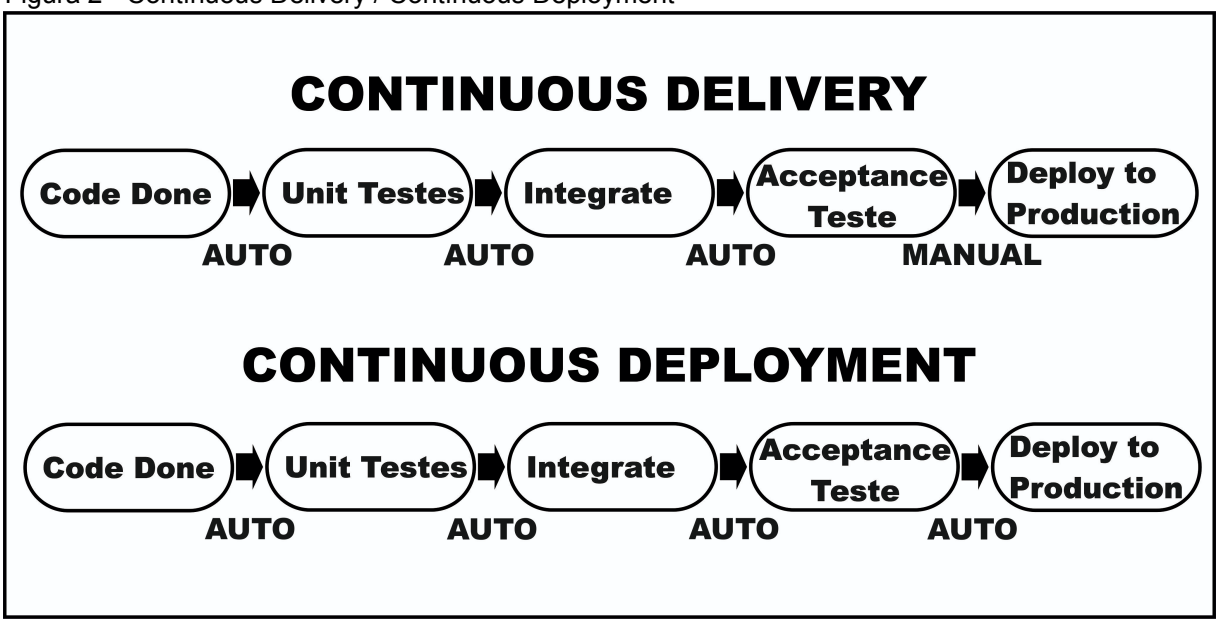
Segundo Azeri (2017), o termo CD pode ser tratado de duas formas. A primeira trata o termo como *Contínuous Delivery* (Entrega Contínua). Esta abordagem visa reduzir os conflitos advindos do processo de implantação do *software* ou liberação de versão para a implantação. Este cenário tem como base uma integração contínua bem estruturada que envolva um alto grau de automação em todas as etapas de desenvolvimento e construção de forma que a liberação de uma *release* segura possa ser realizada a qualquer momento.

A segunda forma trata CD como *Contínuous Deployment* (implantação contínua). Esta abordagem garante o deploy da aplicação de forma automática toda vez que houver uma funcionalidade nova importante no projeto. Este abordagem também precisa ter como base uma integração contínua bem estruturada (AZERI, 2017).

A diferença básica entre as duas abordagens é que na primeira (*Contínuous Delivery*) a *release* fica disponível a qualquer momento para implantação, porém precisa ser implantada manualmente, na segunda abordagem a implantação é feita de forma totalmente automatizada (ver Figura 2) (AZERI, 2017).

Figura 2 - Contínuous Delivery / Contínuous Deployment

Fonte: Adaptado Mandic -



Disponível em: <https://www.mandic.com.br/artigos/explora
ndo-devops-com-foco-em-ci-cd/> Acesso em: 15/08/2018

2.3.3. Controle de Versão

Os sistemas de controle de versão tem a função de registrar e gerenciar as diversas alterações e versões geradas no decorrer da vida útil de um documento ou projeto, oferecendo uma forma inteligente de organizar o código de um projeto de desenvolvimento de *software* (LACERDA, 2012).

Dentre as principais vantagens do uso de sistemas de controle de versões em projetos de *software* está a possibilidade de rastrear mudanças no código, criar versões do código em determinado estado, retornar o código a uma versão estável de desenvolvimento, colaboração paralela, dentre outras vantagens (LACERDA, 2012).

O funcionamento básico dos sistemas de controle de versão se dá através da criação de um repositório (servidor) que armazena o histórico de versões/alterações do código, podendo os desenvolvedores acessarem o repositório fazerem uma cópia do conteúdo atualizado para sua máquina local e atualizarem o conteúdo do servidor com as novas alterações implementadas ao longo do tempo (LACERDA, 2012).

Segundo Dias (2016), os sistemas de controle de versão podem ser divididos em controle de versão distribuído e controle de versão centralizado. O controle de versão centralizado usa a topologia estrela, onde um único repositório central é responsável pelo versionamento do código, sendo que para alterações passar de uma área de trabalho para a outra obrigatoriamente deve passar primeiro pelo servidor. O controle de versão distribuído é composto por vários repositórios locais independentes associados a área de trabalho de cada desenvolvedor, sendo que esses repositórios podem se comunicar entre si atualizando o repositório vizinho ou se atualizando através das operações básicas de *pull* para se atualizar através de outro repositório, e *push* para atualizar outro repositório através de suas próprias alterações.

O SVN *Subversion* é um sistema de controle de versão criado no ano de 2000 pela CollabNet e mantido atualmente pela Apache. O SVN é um dos controles de versão mais conhecidos que se enquadra no tipo centralizado, adotando a arquitetura cliente-servidor (DEVMEDIA, 2014).

O controle de versão centralizado é implantado em um servidor, neste servidor são armazenados e disponibilizados para os clientes (desenvolvedores)

todos arquivos versionados através dos comandos de checkout. Este modelo tem grandes vantagens em relação aos sistemas de versionamento locais, tais como, acesso de múltiplos clientes aos arquivos versionados, rastreabilidade respondendo facilmente as perguntas "quem?", "quando" e "o que alteou?" (GIT, 2018).

Apesar das vantagens acima descritas este sistema tem alguns pontos de desvantagens consideráveis tais como, servidor como ponto de falha. Se uma falha ocorrer no servidor por algumas horas será impossível trabalhar em conjunto ou disponibilizar novas versões. É necessário se prevenir com *backups* diários para mitigar problemas com falhas no disco de armazenamento do servidor (GIT, 2018).

O Git é um sistema de controle de versão distribuído e de código aberto projetado para ser usado em todos os tipos de projetos, grandes ou pequenos, sendo o mais conhecido sistema de controle de versão da categoria distribuído (GIT, 2018).

No sistema de controle de versão distribuído, clientes não fazem meras cópias dos arquivos versionados, mas cada cliente é uma cópia completa do repositório, assim caso o servidor principal falhe pode ser feita a restauração através de um repositório local (GIT).

Outra vantagem importante do controle de versão distribuído é a possibilidade de interação de *push* e *pull* entre repositórios clientes, abrindo um leque de possibilidade de organização dos fluxos de trabalho (*workflow*) que não são possíveis no sistema centralizado (GIT, 2018).

3. MÉTODOS E FERRAMENTAS UTILIZADAS

Conforme o estudo do Capitulo 2 a implementação do CI CD deve seguir alguns requisitos importantes para um bom funcionamento. Neste capitulo será realizada a análise de ferramentas de cada segmento e a escolha de quais irão compor o estudo de caso deste trabalho, tendo como premissa a utilização da linguagem de desenvolvimento Java para plataformas *web*.

3.1. Ferramentas para CI/CD

3.1.1. Repositório de Código

O mundo do desenvolvimento de *software* está cada vez mais consolidando a tendência dos desenvolvedores de *software* utilizarem algum repositório de código fonte, pois estes passam a entender os benefícios desta tecnologia, tais como economia de tempo e compartilhamento de código de forma prática e segura entre os desenvolvedores da equipe (SONATYPE, 2018).

Entre os mais conhecidos repositório de código grátis, além do GitHub que foi adquirido pela Microsoft, estão o GitLab, BitBucket, SourceForge, GitKraken e LaunchPad (ANONYMOUSHACKER, 2018).

Segundo Wayner (2018), os principais repositórios do universo Git são GitHub, GitLab e BitBucket, que tem suas vantagens e desvantagens que norteiam as escolhas dos desenvolvedores de acordo com as suas necessidades, sendo que o foco principal dessas ferramentas é conectar os desenvolvedores para que posam trabalhar no mesmo código com facilidade.

Conforme a analise, o GitHub é o maior entre todos, se considerado o volume de código armazenado. O GitLab e o BitBucket são mais baratos nos planos básicos em relação aos planos básicos do GitHub. O BitBucket e o GitLab permitem a criação ilimitada de repositórios privados e gratuitos para pequenas equipes. O BitBucket e o GitLab dispõem de ferramentas de integração contínua, podendo criar *pipelines* de *build*, testes de implantação e monitoramentos sem sair da ferramenta. Para implementar integração contínua no GitHub é necessário a utilização de ferramentas a parte, chamada de Travis CI. O GitLab conta com um ambiente de desenvolvimento integrado, uma mini IDE para desenvolvimento online, nada

comparado ao eclipse, porém proporciona *commits* limpos de arquivos inúteis. O GitLab é integrado ao Google Cloud facilitando o lançamento de microsserviços através de uma orquestração de contêineres com Kubernets Engine.

3.1.2. Docker

Docker é um *framework* que permite a criação de um ou mais ambientes de desenvolvimento e/ou produção isolados em um mesmo sistema operacional e seu principal foco é a agilidade e facilidade de configuração, sendo muito útil para a automação dos processo de desenvolvimento (RAMOS, 2016).

O Docker vem ganhando o espaço anteriormente ocupado pelas VM's (Máquinas Virtuais), que com o aumento da capacidade dos recursos de *hardware* dos servidores, serviam para dimensionar esses recursos para as aplicações (PAGOTTO, 2017).

Para melhor entender como é o funcionamento do Docker, Pagotto (2017) traça um paralelo entre VM e Docker conforme tratado a seguir.

O aumento da capacidade de armazenamento e processamento de servidores criou um cenário onde as aplicações não conseguiam lidar com a abundância de recursos disponibilizados pelos servidores, deixando muitos recursos de hardware ociosos por longos períodos (PAGOTTO, 2017).

Neste cenário surgiram *softwares* projetados para serem executados entre o hardware e os SO's (Sistemas Operacionais). Esses *softwares* tem o papel de gerenciar os recursos do servidor e criar máquinas virtuais que simulem uma máquina com o *hardware* especifico e dimensionado, com a possibilidade de criar varias máquinas virtuais em um único servidor, desde que o mesmo disponha de recursos suficientes (PAGOTTO, 2017).

Dentro de cada máquina virtual é possível instalar um sistema operacional com *kernel*, binários e bibliotecas exclusivos. Essa característica traz como vantagem, a possibilidade de instalação de diversos sistemas operacionais em um único servidor físico, e uma desvantagem, a instalação de vários sistemas operacionais ocupa um espaço considerável de armazenamento. Outra vantagem que as máquinas virtuais trouxeram foi a redução de custo através de um rápido

provisionamento de infraestrutura e aumento na capacidade de recuperação de falhas catastróficas (PAGOTTO, 2017).

A diferença básica entre máquinas virtuais e contêineres *docker* é o ponto onde é realizada a virtualização, enquanto que as máquinas virtuais virtualizam o *hardware* o *docker* virtualiza o sistema operacional. Os contêineres *docker* criam vários ambientes de execução isolados em apenas um sistema operacional (PAGOTTO, 2017).

Os contêineres ficam acima do *hardware* e acima do sistema operacional do servidor sendo possível a utilização de diversos contêineres que compartilham o *kernel*, binários e bibliotecas do mesmo sistema operacional, porém trabalham de forma isolada em um processo do sistema operacional. Um dos benefícios do contêiner ser executado em um processo é que o processo utiliza os recursos de *hardware* sem virtualização, onde o processo é virtualizado, porém o processamento não (RAMOS, 2016).

Componentes que precisam ser compartilhados entre os contêineres são classificados como somente leitura, contribuindo para que estes se tornem bastante leves em relação as máquinas virtuais. O tamanho médio de um container é estimado em MB e gasta poucos segundos para iniciar, contra alguns minutos e vários GB de uma máquina virtual (PAGOTTO, 2017).

3.1.3. Orquestradores de Contêineres

Os orquestrador de contêineres dão aos gerenciadores de sistemas poderes para gerir um grupo de contêineres necessários para o funcionamento de uma aplicação (TRUCCO, 2017).

Para exemplificar, considerem o cenário hipotético de que para uma aplicação PHP funcionar seja necessária a configuração de um container para o servidor PHP e um para o banco de dados MySQL. Normalmente o profissional responsável pela infraestrutura configura esse ambiente em menos de um dia. Imaginemos que esse *setup* seja necessário para ambientes de desenvolvimento, homologação, produção e deve ser instalado em vários clientes. Este serviço

executado manualmente aumenta a possibilidade de erros, sendo os orquestradores de contêineres os responsáveis por ajudar nessas tarefas (TRUCCO, 2017).

Com os orquestradores é possível reger a integração e o comportamento de diversos contêineres através de arquivos com comandos escrito em linguagem YAML (TRUCCO, 2017).

Segundo o portal stackshare o kubernets é o orquestrador de contêineres mais popular do mundo (STACKSHARE, 2018). O Kubernets inicialmente desenvolvido pela Google foi doado a comunidade se tornando uma ferramenta de código aberto para gerenciamento de contêineres em sistema distribuído, operando em um conjunto de máquinas (cluster) (ELLINGWOOD, 2018).

3.1.4. Ferramentas de Build

Ao terminar a codificação a etapa seguinte é a de testes, porém para testar é necessário a compilação (*build*) do projeto para torná-lo executável e funcional, processo que pode ser realizado manualmente através de ferramentas disponível na sua IDE. A partir do momento que o projeto começa fazer uso de dependências a construção manual se torna onerosa em relação ao tempo gasto na construção do executável, sendo necessário a utilização de ferramentas que automatizem o processo de construção do *software* (MEDEIROS, 2016).

O Maven é uma ferramenta desenvolvida para gerenciar build's de projetos de *software* Java, baseada no conceito POM (*Project Object Model* - Modelo de Objeto de Projeto). Criado com o objetivo de simplificar e padronizar o *build* do projeto Jakarta Turbine, onde a equipe de desenvolvimento dispunha de vários arquivos de compilação sem nenhum padrão. O projeto foi finalizado e o Maven acabou se tornando a ferramenta de build para projetos baseados em Java (APACHE, 2018).

3.1.5. Ambientes de Deploy

O crescimento e a popularização da computação em nuvem facilitou o acesso a serviços disponíveis para *deploy* de aplicações *Web*, desde *WebServices* a aplicações complexas. Dos serviços disponíveis pela computação em nuvem, um dos mais utilizados para pequenos projetos é o que disponibiliza a Plataforma como Serviço (PaaS) (STORI, 2013).

Este serviço utiliza contêineres para abstrair da equipe de desenvolvimento trabalhos relacionados com a configuração de infraestrutura, oferecendo aos desenvolvedores a plataforma pronta para uso dando a vantagem destes dedicarem suas atividades somente ao desenvolvimento do *software*. Ainda entre as vantagens existentes neste serviço está a facilidade de manutenção e escalabilidade além do baixo custo inicial, pois a maioria dos fornecedores do serviço disponibilizam planos gratuitos para projetos que não consumam muitos recursos físicos (STORI, 2013).

Existem atualmente no mercado diversas empresas que oferecem esse serviço, tais como: Heroku, RedHat OpenShift, CloudBees, AWS Elastic Beanstalk, Jelastic, dentre outras. A diferença básica entre cada um está nas tecnologias disponíveis para uso do desenvolvedor, diferença no preço por uso de CPU, memória e disco e facilidade de implantação (deploy) do aplicativo, sendo que neste último requisito o Heroku tem maior destaque (STORI, 2013).

3.2. Ambiente de Desenvolvimento

3.2.1. IDE Java

De acordo com Viswanathan (2018), entre as principais IDE's disponíveis no mercado para Java podemos destacar o Eclipse, NetBeans e IntelliJ IDEA:

O Eclipse foi criado pela IBM em 2001 como uma plataforma de código aberto, e sua principal vantagem é sua capacidade de interação com diferentes tecnologias através da sua sessão de plugins, o que o torna altamente versátil e personalizável.

O NetBeans foi criado no final da década de 90 em uma plataforma de código aberto e posteriormente foi adquirido pela Sun em 1999, hoje, o NetBeans pertence a Oracle. Nesta IDE é possível desenvolver em C / C++ PHP e Java, além de oferecer editores para HTML e XML. Um ponto do NetBeans em cima do Eclipse é a existencia de um front end para os bancos JavaDB, MySQL, PostgreSQL e Oracle.

Criado em 2001 o IntelliJ IDEA conta com uma versão comercial para empresas e uma versão grátis para desenvolvedores individuais, possui suporte para diversas linguagens que incluem Java, possui também uma sessão de plugins onde estes podem ser desenvolvidos pelos usuários, possui uma biblioteca com

cerca de 1000 plugins públicos e cerca de 55 privados disponíveis apenas na versão empresarial.

Para Coman (2014), a escolha entre as três IDE's é muito subjetiva, pois três grandes nomes concorrem para oferecer as melhores funcionalidades, ficando a escolha a depender das necessidades da equipe de desenvolvimento. Para o desenvolvimento do estudo de caso foi utilizado o Eclipse.

3.2.2. Framework Spring Boot

O Spring Boot é um *framework* para aplicações Java derivadas e evoluídas do Spring Framework, e tem como objetivo disponibilizar ferramentas para agilizar o desenvolvimento e facilidade de configuração de aplicações baseadas no Spring Framework (PERRY, 2017).

O *framework* Spring foi desenvolvido inicialmente por Rod Johnson em 2002 para facilitar a construção de aplicações Java, e posteriormente foram sendo adicionados novos modulo, tais como Spring Data para facilitar ao programador desenvolvimento de persistência, Spring Security para tratar a segurança no projeto, dentre outros (GENTIL, 2012).

A medida que o Spring ganhava mais módulos o *framework* se tornava cada vez mais robusto com um conjunto elegante de estruturas que funcionavam perfeitamente, porém com uma configuração complicada onde era necessário muito tempo para iniciar um simples *hello world*, com a configuração de diversos XML's (PERRY, 2017).

O camada Spring Boot veio agregar ao framework Spring a facilidade de configuração, que atualmente é praticamente zero , sendo capaz de configurar automaticamente na inicialização do projeto os XML's e os data sources, dando aos programadores a possibilidade de focar no desenvolvimento usufruindo de todos os benefícios do *framework* Spring de forma rápida (SANTANA, 2016).

Segundo Perry (2017), um dos pontos mais importantes a serem entendidos no Spring Boot são os inicializadores, que são um conjunto de dependências responsáveis cuja responsabilidade é reduzir a configuração manual. Por padrão os inicializadores tem o nome padrão de spring-boot-starter-XYZ, onde XYZ é o nome

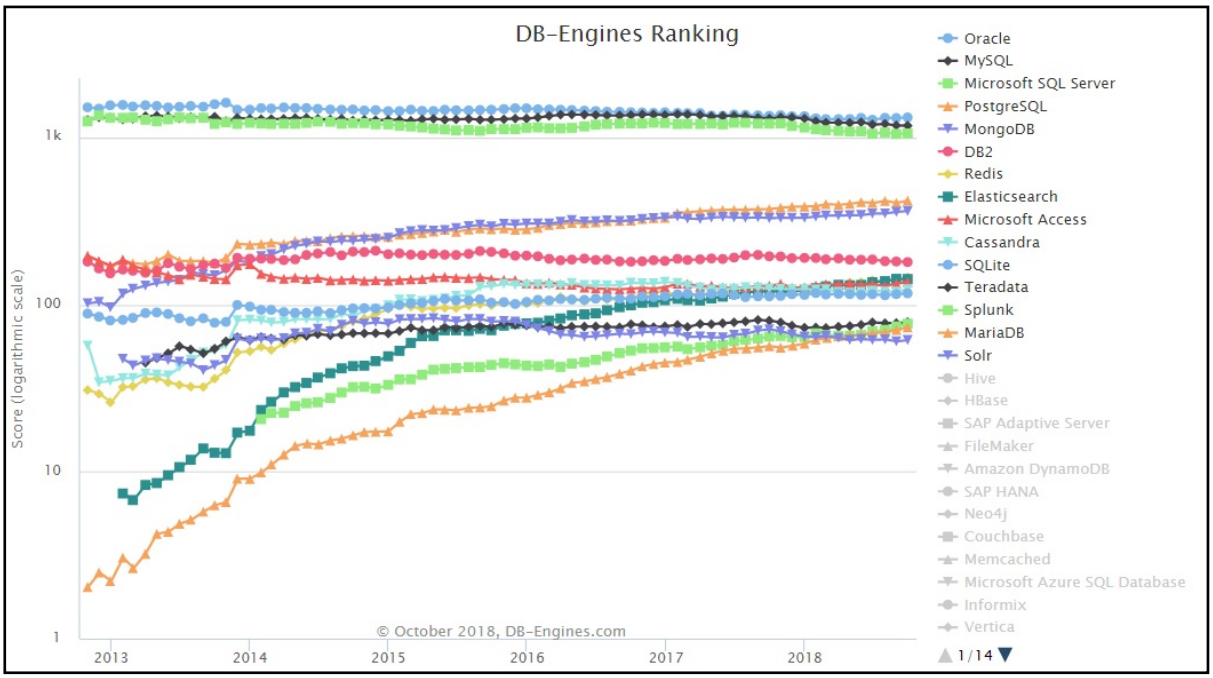
do tipo de aplicação que se deseja desenvolver. Dentre os principais inicializadores está o spring-boot-starter-web, para desenvolvimento aplicações web com Spring MVC e Tomcat embarcado, e o spring-boot-starter-jersey, que é semelhante ao web, porém usa o Apache Jersey no lugar do Spring MVC.

Dentre as principais vantagens do Spring Boot está a rápida inicialização do Spring Boot, construção rápida de REST API, Web Socket, Web dentre outros, suporte a SQL e NoSql, servidores tomcat embarcado, reinicio automatico do servidor ao salvar alterações no código, inclusão fácil e rápida de dependências prontas para serem utilizadas (SPRING.IO, 2018).

3.2.3. Sistema Gerenciador de Banco de Dados

A empresa austríaca Solid IT criou em 2002 um portal chamado DB-Engines para ranquear os melhores sistemas gerenciadores de banco de dados disponíveis no mercado, e desde 2014 os 5 SGBD's mais utilizados na sequência de popularidade são Oracle, MySQL, Microsoft SQL Server, PostgreSQL e MongoDB como podemos visualizar na imagem da figura 3 a seguir (DB-ENGINES, 2018).

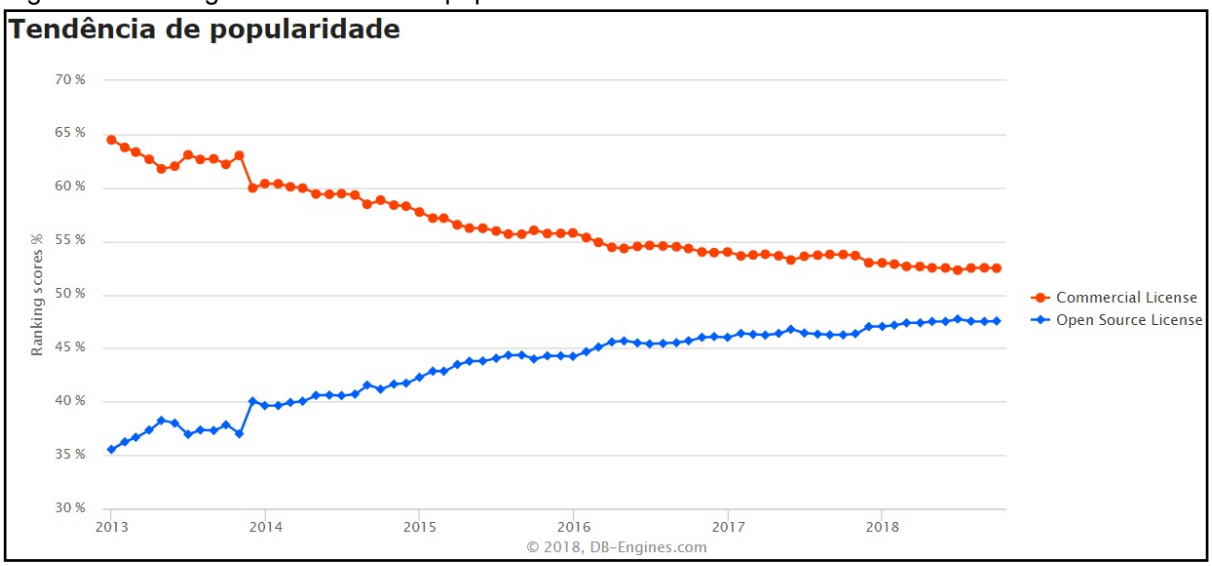
Figura 3 - DB-Engines Ranking



Fonte: DB-Engines - Disponível em <https://db-engines.com/en/> Acesso em: 18/08/2018

O portal conta com 346 sistemas catalogados em setembro de 2018, desses 50% são sistemas de código aberto e 50% sistemas comerciais pagos. Em comparação ao nível de popularidade os sistemas open sources seguram 47,5% do interesse de uso e com uma tendência de aumentar o nível de popularidade, podendo até mesmo ultrapassar os sistemas comerciais nos próximos anos como podemos verificar na figura 4.

Figura 4 - DB-Engines Tendência de popularidade



Fonte: DB-Engines - Disponível em <https://db-engines.com/en/> Acesso em 18/08/2018

Dos 5 sistemas do topo da classificação, 3 são sistemas de código aberto, sendo eles, MySQL, PostgreSQL e MongoDB dos quais serão tratados a seguir.

O MongoDB é um banco de dados orientado a documentos, onde cada registro armazena todas as informações necessárias em um único documento. Por não utilizar SQL os bancos de dados orientados a documentos são conhecidos como bancos de dados NoSQL. Dentre as suas principais vantagens podemos destacar a performance, pois uma única consulta retorna todas as informações necessárias do documento, e a capacidade de lidar com grande volumes de dados muito utilizados no campo do BigData (MEDEIROS, 2014).

O MySQL é um SGBD relacional, criado em 1996 pela empresa TcX com o intuito de prover um banco de dados relacional simples, rápido, de baixo custo e que rodasse em hardwares baratos (SANTOS, 2006). Utiliza o SQL como interface e foi

desenvolvido para projetos de pequeno a médio porte, inicialmente com carga máxima de 100 milhões de registros por tabela, o que foi sendo melhorado em suas novas versões aumentando o limite de sua capacidade. Pode-se destacar como sua principal vantagem o seu desempenho, com operações extremamente rápidas, o que o torna favorito para sistemas web com um grande volume de transações (TEIXEIRA, 2013).

O PostgreSQL é um SGBD relacional concebido em 1986 pelo departamento de ciência da computação de Berkely University of California, sendo o primeiro SGBD a apresentar vários conceitos importantes no mundo dos bancos de dados relacionais como chaves estrangeiras, functions, triggers, Views, que só vieram a se consolidar nos SGBD's comerciais muitos anos depois (DIONISIO, 2015). O PostrgreSQL não tem limite quanto ao tamanho de banco de dados que ele possa gerenciar, tendo apenas o limite físico de hardware. Em nível de tabelas, o PostrgreSQL pode manter até 32TB, sendo que cada registro pode ter até 1,6TB com até 1GB por cada campo. É possível ainda ter índices ilimitados para acelerar a busca de dados (MILANI, 2008).

O MongoDB foi descartado da possibilidade de uso no caso de uso por ser uma tecnologia voltada para o gerenciamento de grandes massas de dados, o que não é o caso do nosso estudo. O MySQL apresenta como principal vantagem a agilidade nas transações, porém não se mostra tão robusto quanto o PostgreSQL, levando à escolha do PostgreSQL.

3.2.4. Framework Front End

Os frameworks front-end são ferramentas desenvolvidas para auxiliar a criação de interfaces de um sistema para o usuário, sendo indispensável para sistemas web que tem grande interação com os usuários (SCUDERO, 2018).

Segundo Scudero (2018), uma empresa Suíça realizou um levantamento e ranqueou as seis melhores ferramentas para auxílio de criação de interfaces web ficando classificados do menos utilizado para o mais utilizados os seguintes frameworks: Pure, Material UI, Semantic UI, Materialize, Foundation, Bootstrap.

Foi escolhido para a implementação das interfaces o Materialize que faz uso da tecnologia JQuery, o que agrega diversos recursos visuais, além de conter elementos visuais para dispositivos móveis

3.2.5. Metodologia de Desenvolvimento

Inicialmente foi cogitada a possibilidade de utilizar a metodologia ágil XP para o desenvolvimento do projeto, visto que um dos conceitos estudados neste trabalho teve origem no XP, onde uma de suas doze práticas orienta a prática da integração contínua que consiste em agregar com frequência as novas funcionalidades desenvolvidas ao projeto principal.

Posteriormente ao estudar sobre integração contínua Fowler (2006) frisa que um dos requisitos essenciais para a implementação do conceito é que o projeto tenha um alto grau de testes automatizados no código, o que reforçava ainda mais a convicção do autor a utilização do XP, pois, uma das práticas do XP é a elaboração de um teste para cada funcionalidade.

A prática de elaborar um teste para cada funcionalidade se mostrou tão eficaz no XP que deu origem a uma nova metodologia de desenvolvimento, o TDD, que como estudado, realiza o planejamento das interfaces, codifica os testes, implementa a funcionalidade e refatora o código finalizando o ciclo, que se repete para cada funcionalidade.

A negligência na elaboração e desenvolvimento de testes por parte dos programadores já era observada por Rocha (2013), onde o mesmo propunha a adoção de boas práticas à metodologia para evitar esse problema.

Considerando o fato de testes unitários automatizados serem essenciais na implementação dos conceitos de integração contínua e ser um ponto muito negligenciado no processo de desenvolvimento será usada a metodologia TDD, que garante os testes unitários, faltando apenas a automação para atender este requisito do conceito.

3.2.6 Aplicação do Framework XUnit

Para auxiliar a elaboração de testes proposta pela metodologia de desenvolvimento iremos utilizar o framework de teste XUnits em sua versão para Java como foi proposto por Rocha.

O JUnit é uma variante do framework XUnit desenvolvido para a linguagem Java e tem a função de facilitar a criação e execução de testes em um projeto Java. Com os testes elaborados e implementados com a API do JUnit é possível verificar se cada unidade de código apresenta o comportamento esperado, além de ter recursos que facilitam a execução automática de todos os testes e a exibição dos resultados (MEDEIROS, 2009).

3.3. Integração Contínua

Como estudado no tópico 2.3.1. deste trabalho o conceito de integração contínua tem o objetivo de integrar rapidamente novas funcionalidades desenvolvidas ao projeto principal com o menor risco de problemas no código, e para implementar isso é importante a utilização de algumas tecnologias paralelas que em conjunto atingem o objetivo. Essas tecnologias são: Repositório de Código, Sistemas de Controle de Versão e uma ferramenta de Integração contínua com capacidade de automatizar o build e testes unitários.

De acordo com estudo realizado no tópico 2.3.3. deste trabalho podemos separar os sistemas de controle de versão em duas partes, centralizado e distribuído, onde podemos destacar o SVN como sistema centralizado e o Git como o sistema distribuído. No estudo do capítulo pudemos concluir que os sistemas de controle de versão distribuídos tem algumas vantagens em relação aos sistemas de controle de versão centralizados, onde temos repositórios completos e independentes em cada estação de trabalho no sistema distribuído. Essas vantagens levaram o autor a uma busca por repositórios que usam a tecnologia Git como sistema de versionamento.

De acordo com o estudo do tópico 3.1.1. dentre os principais repositórios que trabalham em projetos versionados pelo Git podemos destacar o GitHub, GitLab e Bitbucket. Ao realizar a consulta no site oficial de cada repositório foram extraídas as informações da Figura 5.

Figura 5 - Analise Comparativa Repositório de Código

Plano Grátis	GitHub	GitLab	Bitbucket
Repositórios públicos	Ilimitado	Ilimitado	Ilimitado
Repositórios Privados	0	Ilimitado	Ilimitado
Colaboradores por projeto	Ilimitado para repositório público	Ilimitado	5 por projeto
Espaço armazenamento	--	10GB	1GB
Plano pago mais barato	US $ 7	US $ 4	US $ 2
Integração Contínua própria	Ferramentas de terceiros - Travis CI	Ferramenta Própria - GitLab CI	Ferramentas de terceiros - JIRA

Fonte: O Autor.

Como podemos observar, o GitHub oferece repositórios ilimitados para projetos públicos de código aberto, com vantagem para o GitLab e o Bitbucket neste quesito, que oferece repositórios públicos e privados ilimitados. Para os repositórios privados o GitLab leva vantagem sobre o Bitbucket na quantidade de colaboradores por projeto, que são ilimitados para o GitLab e 5 para o Bitbucket. O GitLab disponibiliza 10GB para repositórios privados e o Bitbucket 1GB. Se comparado os preços dos planos pagos dos três repositórios o mais caro é o GitHub, seguido do GitLab e Bitbucket. O GitLab tem uma ferramenta própria de integração contínua dispensando configurações para integrar a uma ferramenta de terceiros como o GitHub e Bitbucket.

Para o desenvolvimento do conceito foi escolhido o GitLab.

3.4. Deploy Contínuo

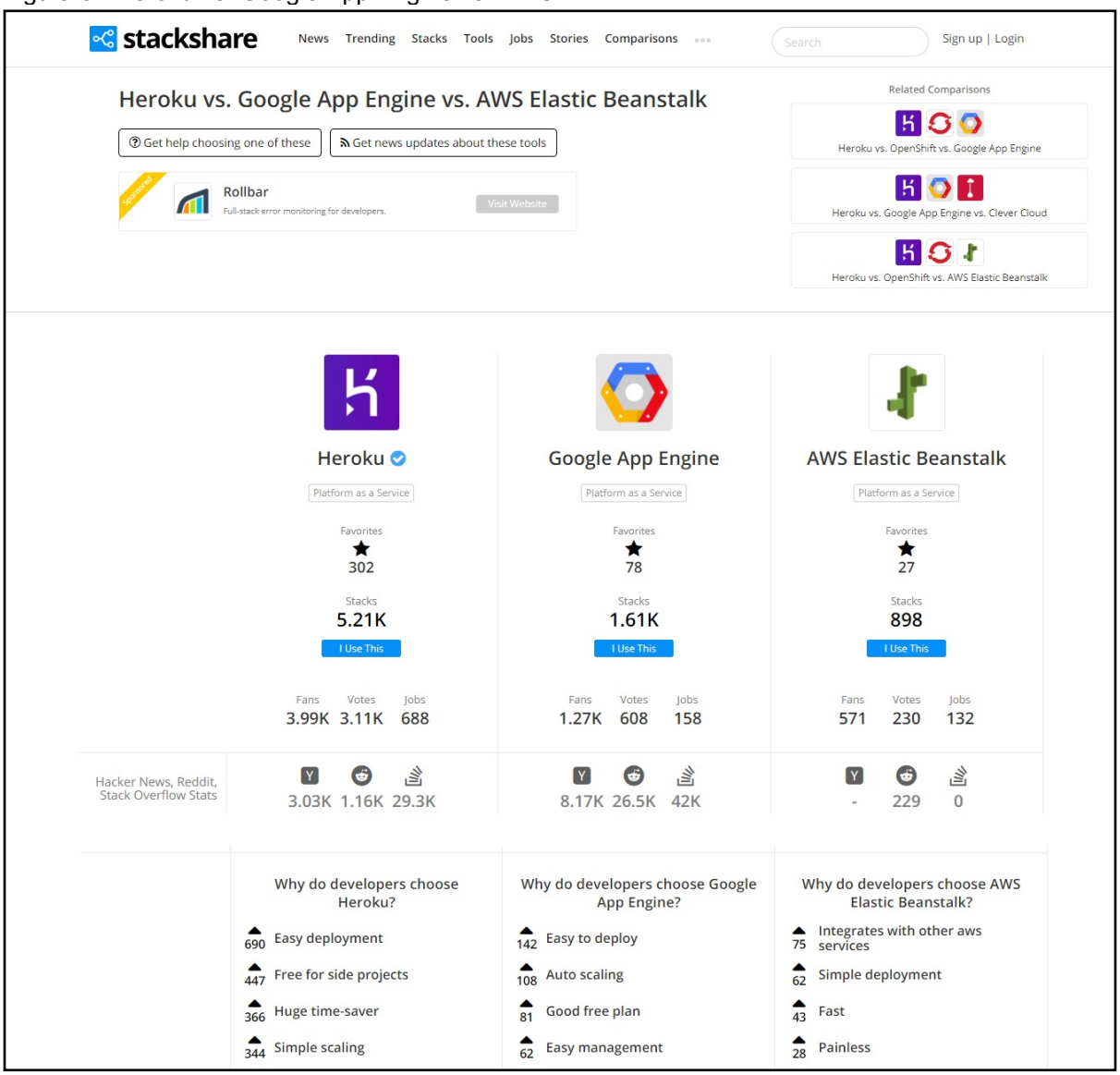
Como foi estudado no tópico 3.1.5. deste trabalho, percebemos a importância da tecnologia PaaS (Plataforma como serviço) para o deploy de aplicações.

O Portal Stackshare.io tem como missão a seguinte frase "We're *on a mission to transform the way that all SaaS tools and infrastructure are bought and sold*", que em tradução livre significa "Temos a missão de transformar a forma de vender tecnologias SaaS e infraestruturas". Este portal basicamente elabora ranques

de diversas tecnologias com base nas interações da comunidade, funcionando como uma espécie de rede social das tecnologias (STACKSHARE, 2018).

O ranque PaaS, apresenta um comparativo entre os três primeiros colocados, relacionados do mais popular ao menos popular: Heroku, Google App Engine e AWS Elastic Beanstalk (STACKSHARE, 2018). Veja figura 6.

Figura 6 - Heroku vs. Google App Engine vs. AWS



Fonte: StackShare - Disponível em:<https://stackshare.io/stackups/aws-elastic-beanstalk-vs-google-app-engine-vs-heroku> Acesso em 20/08/2018.

O Heroku é uma plataforma de aplicações web que facilita a implantação dos aplicativos, dando aos programadores a possibilidade de usar 100% do tempo desenvolvendo códigos sem a preocupação de gerenciar servidores de implantação.

Dentre os principais motivos para a escolha do Heroku como plataforma de aplicativos estão: fácil implantação, grátis para projetos paralelos, economia de tempo, requer pouco conhecimento *DevOps*, fácil configuração, complementos para diversas tecnologias, curva de aprendizado baixa, dentre outras vantagens (STACKSHARE, 2018).

Com uma reputação altamente confiável o Google App Engine aproveita os 10 anos de experiência do Google com sistemas escalonáveis para criar uma ferramenta de criação de aplicativos fáceis de manter e dimensionar de acordo com o tráfego de dados. Dentre as principais vantagens levantadas pelos usuários estão: Facilidade de implantação, escalabilidade automática, bom plano grátis, baixo custo, conjunto abrangente de recursos, dentre outros (STACKSHARE, 2018).

O AWS Elastic Beanstalk é uma ferramenta da AWS para implementar e gerenciar aplicativos na nuvem da AWS, onde depois de realizar o *upload*, o Elastic provisiona automaticamente a implantação, o balanceamento de carga, dimensionamento e monitoramento da integridade do aplicativo. Dentre as principais vantagens estão: integração com outros serviços AWS e Implantação simples (STACKSHARE, 2018).

Com base no *feedback* dos usuários apontados no portal Stackshare.io a ferramenta escolhida para deploy do nosso estudo de caso será o Heroku. Ao concluir a implementação do *Contínuous Delivery* e *Contínuous Deploy* no Heroku serão exploradas as possibilidades de uso das ferramentas do Google App Engine, visto que é uma plataforma paga que oferece U$ 300,00, aproximadamente R$ 1.100,00, para testar as ferramentas da plataforma.

4. PROJETO DE SOFTWARE

4.1. DESCRIÇÃO DO SISTEMA

O Hospital Alpha é um hospital especialista nos tratamentos de problemas da visão, que muitas vezes precisa de intervenção cirúrgica para obter um bom prognóstico. O Hospital Alpha realiza cirurgias através de vários convênios, porém muitos procedimentos cirúrgicos não tem cobertura nos planos de saúde, sendo estes procedimentos pagos pelo próprio paciente. Quando um paciente é indicado a realizar um procedimento cirúrgico, este é encaminhado a um departamento do hospital especializado em fornecer orientações pré e pós operatórias, além de apresentar ao paciente um orçamento financeiro da cirurgia.

Estes orçamentos podem ter valores diferentes para um mesmo procedimento, pois pode variar a técnica cirúrgica e/ou os materiais e medicamentos usados na cirurgia a depender das condições fisiológicas do paciente, ficando a escolha por conta do paciente de acordo com suas condições financeiras. Um orçamento proposto pode ser aceito, pode ser escolhido uma configuração cirúrgica mais barata, pode ser deixado em stand-by para o paciente se adequar financeiramente ou pode ser recusado.

O sistema de gestão do hospital não disponibiliza uma ferramenta para gestão dos orçamentos, sendo necessário a equipe do departamento fazer a gestão através de uma planilha compartilhada na rede como podemos ver na figura 7.

Figura 7 - Orçamento Cirúrgico

Orçamento

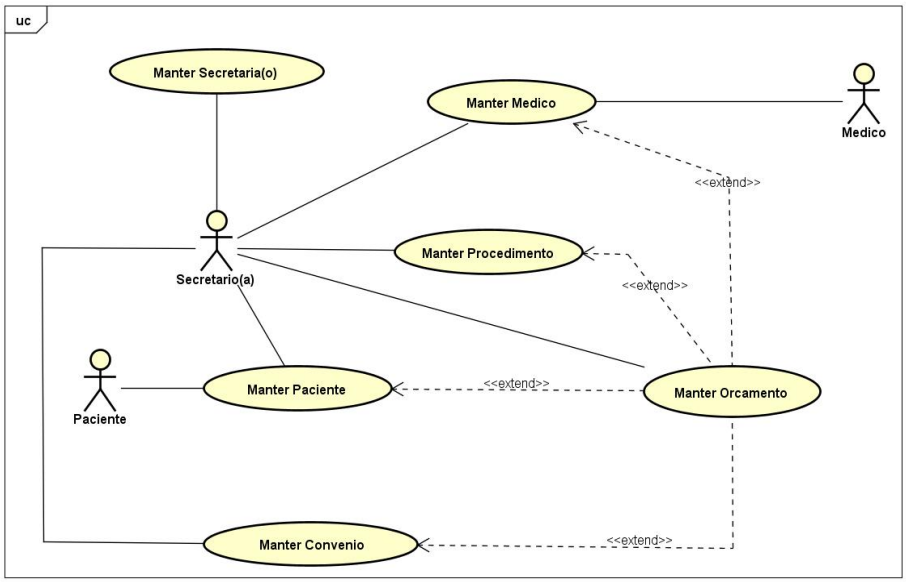
Nº	Nome	Médico	Indicaçã	t. Orçan	Funcio	Hora	Cirurgia	Convenio	Orçamento
1	DA SILVA	NA		10/3/16		10:05:00	CAPSULOTOMIA YAG LASER AO	IPASGO	
2	LEITE	ENA	IHOA	10/3/16		10:20:00	TU PALPEBRAL OD	UNIMED	
1	A BAHIA	ENA	IHOA	10/3/16		AR 10:50:00	PTERIGEO OD	UNIMED	
2	IRA MORAES	MORIM		10/3/16		AR 10:55:00	RET. CE CA / OE	UNIMED	
3	DE ARAUJO			10/3/16		AR 10:55:00	LUCENTIS AO	BASE AEREA	
1	GOLVEIA			10/3/16		AR 11:00:00	AVASTIN OD	PART	1.500,00
2	ES			10/3/16		AR 15:00:00	DERMATO/SUP/INF	IPASGO=PART	3.500,000+250,00 6xcc
4	E DIAS DE CARVALHO		EDUARDO	10/3/16		AR 17:00:00	REC. PÁLPEBRA+SUT DE CONJUNTIVA	BASE AÉREA	650,00 ANESTESIA
3	BORGES			10/3/16		AR 17:10:00	DERMATO/SUP/INF	IPASGO=PART	3.500,000+250,00
4	BORGES			10/3/16		AR 17:10:00	BOTOX	PART	1.400,00 1 AMPOLA E 1/2AMPOLA 950,00
3	ELO CAMARGO			10/3/16		13:45:00	YAG AO	IPASGO	
4	OCHA			10/3/16		14:00:00	YAG OE	IPASGO	
1	CAVALCANTE			10/3/16		15:00:00	EXP/IMP + VITRE ANT + CAPSULO O	SAUDE CX	1.200,00 lio sensar/ plano paga 600,00
5	DOS SANTOS			10/3/16		15:37:00	YAG AO	IPASGO	
2	ES DE ASSIS			10/3/16		16:25:00	TUMOR PALPEBRAL OE	PART	1.800,00 3X OU 1.500,00AV + 100,00ANATOMO
3	LHO CAMARGO			10/3/16		17:18:00	DERMATO SUP	IPASGO/PART	1.950,00 1+3 CC OU 1.800,00AV + 150,00ANEST AV
6	DA DA SILVA	NA	IHOA	10/4/16		09:44:00	YAG AO	IPASGO	
5	GONÇALVES MARTINS			10/4/16		AR 09:05:00	CALAZIO	BASE AEREA	
7	RICO XAVIER DOS SANTOS			10/4/16		10:00:00	PTERIGIO OE	IPASGO	

Fonte: O Autor.

Para solucionar o problema foi proposto o desenvolvimento de um sistema Web para gerir essas informações com os seguintes requisitos: O secretário deve registrar no sistema um orçamento contendo os dados do paciente, valor negociado com o paciente, qual médico realizou a indicação cirúrgica, data atual, qual procedimento cirúrgico está sendo orçado e o convênio do paciente; O secretário deve poder cadastrar outros secretários, bem como médicos e procedimentos.

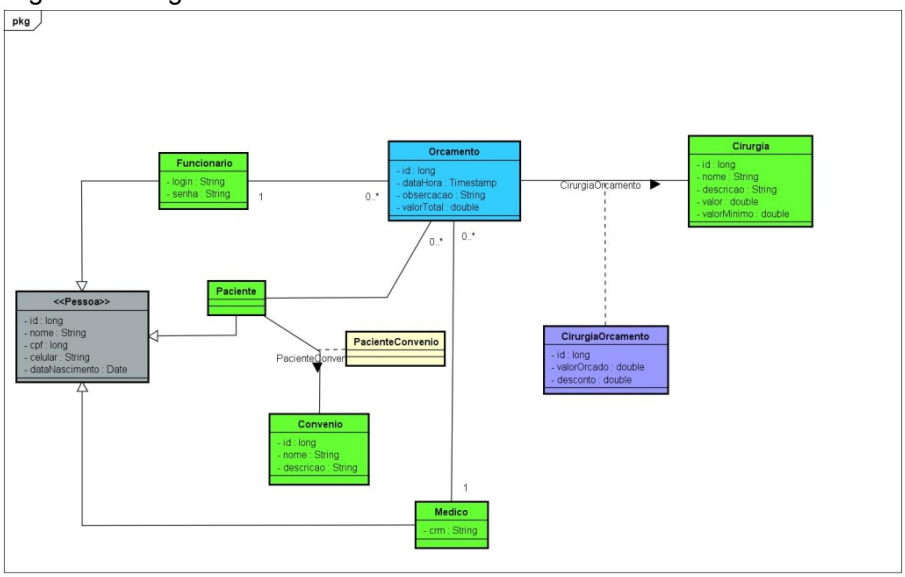
4.2. Diagrama de Caso de Uso / Diagrama de Classes

Figura 8 - Diagrama de Caso de Uso



Fonte: O Autor.

Figura 9 - Diagrama de Classes



Ver Anexos.

O diagrama de Caso de uso foi elaborado com três atores, sendo o secretário

o ator que irá utilizar todo o sistema, os atores paciente e médico são apenas fornecedores de informação ao sistema. Os casos de uso são: Manter Secretario, Manter Médico, Manter Procedimento, Manter Paciente, Manter Convênio e o principal Manter Orçamento. Ver Figura 8.

Na estruturação das classes foram criadas as classes Funcionários, Paciente e Médicos estendidas da classe genérica Pessoa, foram criadas as classes Convênio e Cirurgia para auxiliar a classe principal Orçamento. Ver Figura 9.

4.3. A IMPLEMENTAÇÃO DO SOFTWARE

O inicio do desenvolvimento da aplicação se deu antes do estudo bibliográfico, e surgiu como um projeto paralelo do autor, onde foram levantados os requisitos, elaborado o diagrama de caso de uso, elaborado o diagrama de classes, iniciado o desenvolvimento *Web* utilizando Java com o framework Spring Boot seguindo o modelo arquitetural MVC e utilizando o GitLab como repositório de código. Nesta momento não foi aplicado nenhuma metodologia de desenvolvimento e/ou ferramentas com o intuito de aplicar CI/CD.

Ao iniciar o estudo bibliográfico percebeu-se que o projeto em desenvolvimento, corrigindo algumas deficiências, poderia ser utilizado como caso de uso para aplicação dos conceitos de integração contínua e deploy contínuo.

O desenvolvimento foi parcialmente reiniciado para utilizar a metodologia de desenvolvimento TDD com o *framework* de testes JUnit, visto que sem os testes unitários não seria possível aplicar a integração contínua. Do antigo desenvolvimento foram aproveitados somente as classes da camada Modelo da arquitetura MVC.

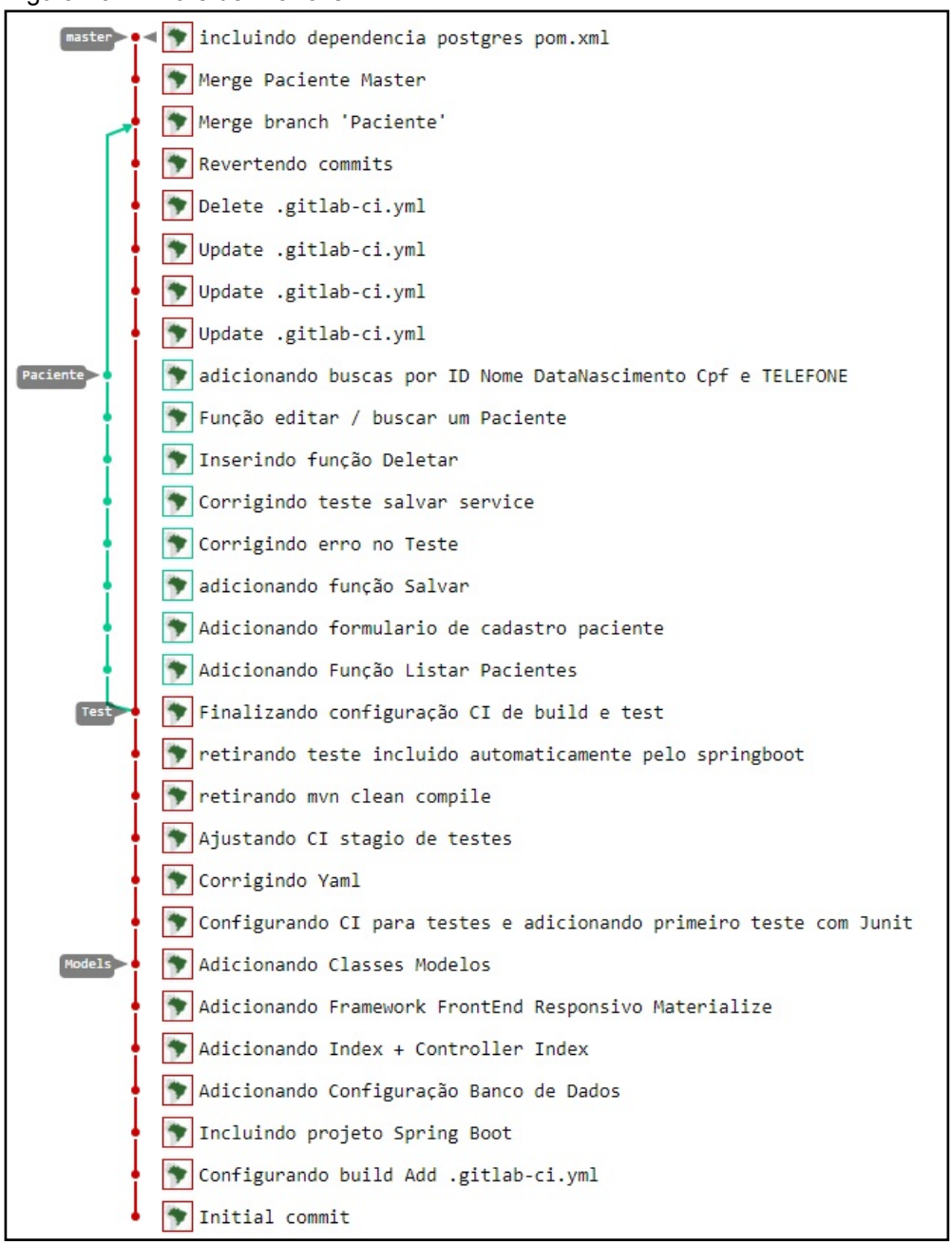
O desenvolvimento se deu com a inclusão de um projeto SpringBoot limpo que posteriormente foi configurado para utilizar o PostgreSQL como banco de dados. Neste momento foi iniciado o controle de versões com o Git localmente, além de realizar o push dos commits para o repositorio de código GitLab. Ainda antes do inicio do desenvolvimento das funcionalidades foi adicionado também o framework FrontEnd responsivo Materialize para melhorar visualmente as view's HTML geradas como interface. Neste ponto do projeto foram adicionados as classes do modelo do

projeto anterior como podemos observar na figura 10 que mostra a arvore de branchs criadas no projeto.

4.3.1. Uso do GitLab CI

Neste ponto do projeto foi onde realmente começou a utilização da ferramenta de integração contínua do GitLab, o GitLab CI, onde foi adicionado o arquivo .gitlab-ci.yml para criar uma pipeline de testes e uma de build do projeto, e paralelamente a inclusão do *framework* de teste Junit no código para a elaboração dos testes unitários. Com as pipelines criadas o GitLab CI fica responsável por executá-las sempre que a *branch* receber uma alteração.

Figura 10 - Arvore de Branchs



Fonte: O Autor.

O arquivo .gitlab-ci.yml deve ser adicionado à raiz do projeto com instruções de como o GitLab CI deve se comportar.

Inicialmente é preciso informar ao GitLab uma imagem *docker* do repositório de imagens Docker Hub que irá rodar os estágios/pipelines. Foi utilizada a imagem do maven:3-jdk-8 que é uma imagem mista de maven 3 com JDK 8. Na sequência é necessário definir os estágios necessários ao projeto, onde foram definidos os estágios de build e testes como podemos ver na figura 11. Com essas configurações o Git Lab está apto a realizar a construção e os testes unitários automatizados toda vez que houver uma alteração no código fonte do projeto.

Fonte: O Autor.

Figura 11 - .gitlab-ci.yaml

.gitlab-ci.yml 167 Bytes

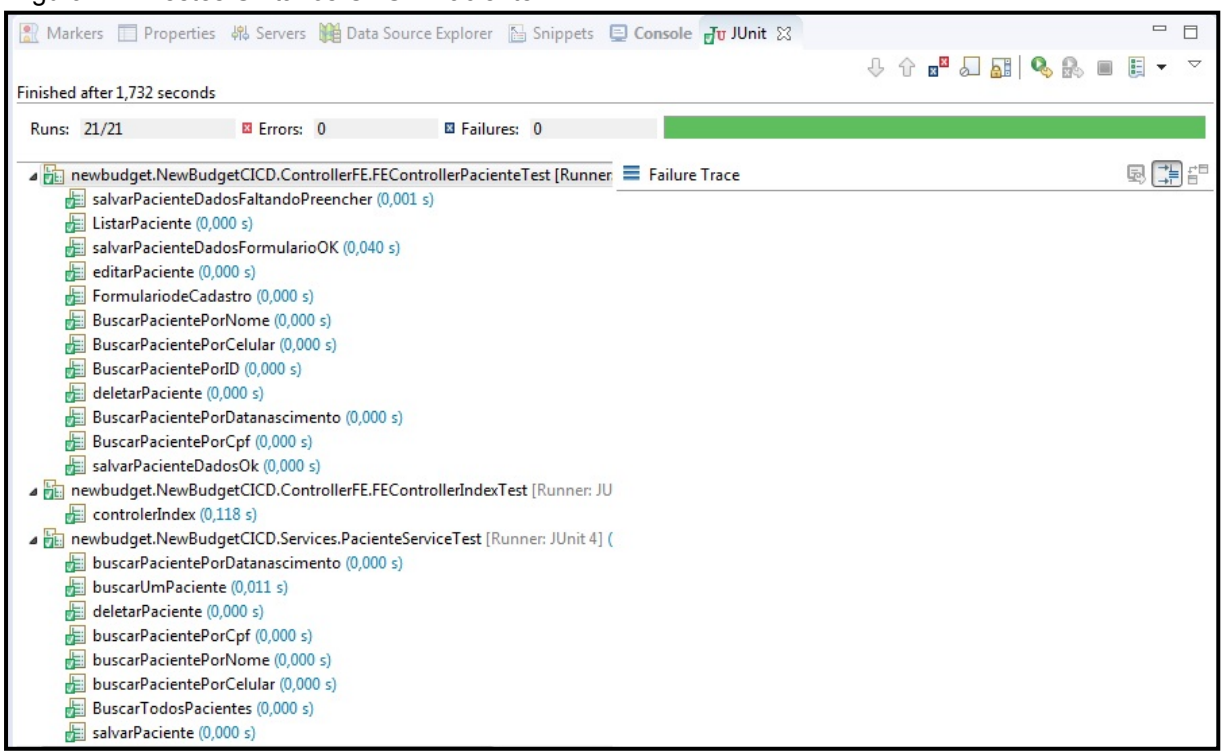
```
1  image: maven:3-jdk-8
2
3  stages:
4      - build
5      - test
6
7  build:
8      stage: build
9      script:
10       - "mvn compile"
11
12 test:
13     stage: test
14     script:
15         - "mvn test"
```

4.3.2. Exploração dos Testes de Softwares

Após a configuração da integração contínua o projeto foi ramificado para a branch Paciente, onde iniciou-se o desenvolvimento do CRUD Paciente com a metodologia TDD, onde para cada funcionalidade desenvolvida eram executados todos os testes localmente e posteriormente enviada para o repositório de código que automaticamente executava as pipelines do projeto.

No Desenvolvimento do CRUD Paciente foram desenvolvidos 20 testes unitários separados entre a camada *Controller* e a camada *Services* como podemos observar na Figura 12.

Figura 12 - Testes Unitários CRUD Paciente



Fonte: O Autor.

Durante o desenvolvimento ao finalizar a função salvar a mesma apresentou problemas, ao detectar o erro na execução da estágio de testes o gitlab imediatamente disparou um email de alerta para correção do código além de apresentar o estagio com falha como mostra a figura 13.

Figura 13 - Pipeline Failed

Fonte: O Autor.

Finalizado o desenvolvimento do CRUD foi realizado o merge da *branch* Paciente com a *branch master* sem intercorrências como podemos observar na figura 14.

Figura 14 - Pipeline Passed

Fonte: O Autor.

4.3.3. Entrega Contínua - Deploy Manual

Concluída a integração contínua foi dado inicio a implementação do Contínuous Delivery/Deploy com a ferramenta Heroku. Inicialmente foi implementado o Contínuous Delivery, que consiste no deploy manual da aplicação após feedback positivo da ferramenta de integração contínua.

O Heroku disponibiliza uma interface de linha de comando, o Heroku CLI, para a configuração e deploy da aplicação, que para funcionar corretamente, precisa realizar uma autenticação, sendo necessário estar devidamente cadastrado no heroku. Após a autenticação foi criada via linha de comando uma aplicação chamada newbudgetcicd e provisionado à aplicação um banco de dados postgresql da camada grátis disponível pelo heroku. Os comandos executados para o deploy manual podem ser vistos nas figuras 15 e 16.

Figura 15 - Deploy Heroku 1

Fonte: O Autor.

```
Administrador: Windows PowerShell
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Todos os direitos reservados.

PS C:\Users\Administrador> cd..
PS C:\Users> cd..
PS C:\> cd newbudgetcicd
PS C:\newbudgetcicd> heroku login
heroku: Enter your login credentials
Email [hefrayn.eu@gmail.com]: hefrayn.eu@gmail.com
Password: ****************
Logged in as hefrayn.eu@gmail.com
PS C:\newbudgetcicd> heroku create newbudgetcicd
Creating                  ... done
https://newbudgetcicd.herokuapp.com/ | https://git.heroku.com/newbudgetcicd.git
PS C:\newbudgetcicd> git push heroku master
Enumerating objects: 136, done.
Counting objects: 100% (122/122), done.
Delta compression using up to 2 threads
Compressing objects: 100% (87/87), done.
Writing objects: 100% (98/98), 12.23 KiB | 736.00 KiB/s, done.
Total 98 (delta 37), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Java app detected
remote: -----> Installing JDK 1.8... done
remote: -----> Executing: ./mvnw -DskipTests clean dependency:list install
remote:        /tmp/build_3b5e2f1d930ff889bf28d171ee518e0a
remote:        [INFO] Scanning for projects...
remote:        [WARNING]
remote:        [WARNING] Some problems were encountered while building the effective model for newbudget:NewBudgetCICD:j
ar:0.0.1-SNAPSHOT
remote:        [WARNING] 'dependencies.dependency.<groupId:artifactId:type:classifier>' must be unique: org.postgresql:p
ostgresql:jar -> version (?) vs 42.2.1 @ line 46, column 15
remote:        [WARNING]
remote:        [WARNING] It is highly recommended to fix these problems because they threaten the stability of your buil
d.
remote:        [WARNING]
remote:        [WARNING] For this reason, future Maven versions might no longer support building such malformed projects
.
remote:        [WARNING]
remote:        [INFO]
remote:        [INFO] -----------------------< newbudget:NewBudgetCICD >-----------------------
remote:        [INFO] Building NewBudgetCICD 0.0.1-SNAPSHOT
remote:        [INFO] --------------------------------[ jar ]---------------------------------
remote:        [INFO]
remote:        [INFO] --- maven-clean-plugin:3.0.0:clean (default-clean) @ NewBudgetCICD ---
remote:        [INFO]
remote:        [INFO] --- maven-dependency-plugin:3.0.2:list (default-cli) @ NewBudgetCICD ---
remote:        [INFO]
```

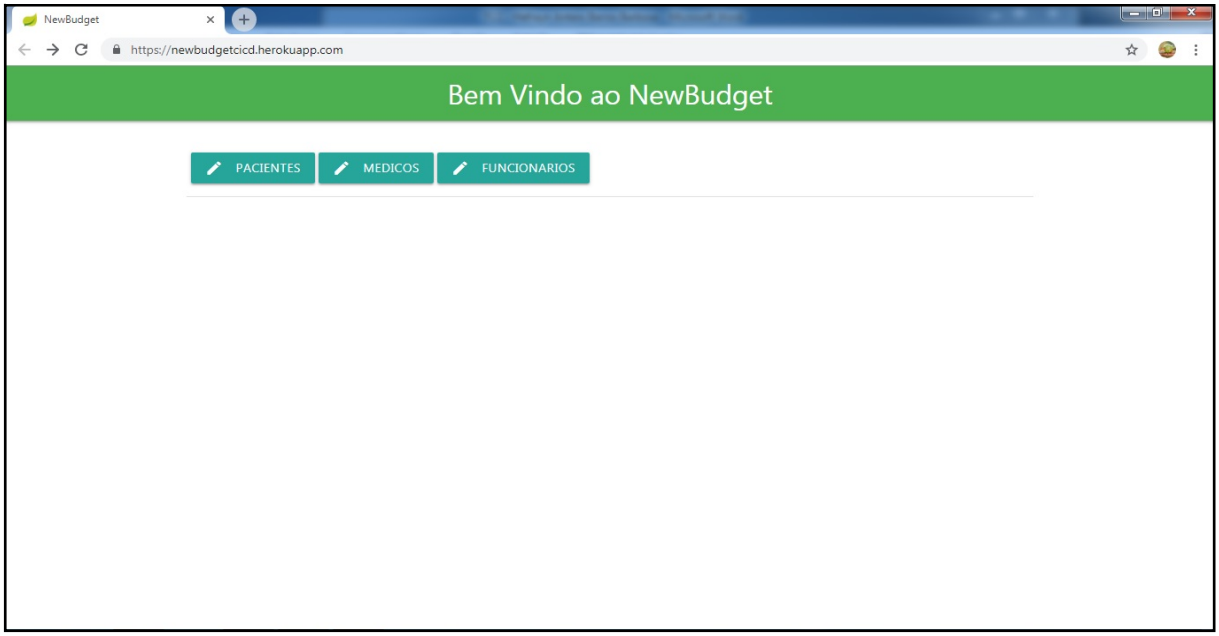
Figura 16 - Deploy Heroku 2

```
Administrador: Windows PowerShell
remote:        [INFO]
remote:        [INFO] --- maven-compiler-plugin:3.7.0:testCompile (default-testCompile) @ NewBudgetCICD ---
remote:        [INFO] Changes detected - recompiling the module!
remote:        [INFO] Compiling 5 source files to /tmp/build_3b5e2f1d930ff889bf28d171ee518e0a/target/test-classes
remote:        [INFO]
remote:        [INFO] --- maven-surefire-plugin:2.21.0:test (default-test) @ NewBudgetCICD ---
remote:        [INFO] Tests are skipped.
remote:        [INFO]
remote:        [INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ NewBudgetCICD ---
remote:        [INFO] Building jar: /tmp/build_3b5e2f1d930ff889bf28d171ee518e0a/target/NewBudgetCICD-0.0.1-SNAPSHOT.jar
remote:        [INFO]
remote:        [INFO] --- spring-boot-maven-plugin:2.0.4.RELEASE:repackage (default) @ NewBudgetCICD ---
remote:        [INFO]
remote:        [INFO] --- maven-install-plugin:2.5.2:install (default-install) @ NewBudgetCICD ---
remote:        [INFO] Installing /tmp/build_3b5e2f1d930ff889bf28d171ee518e0a/target/NewBudgetCICD-0.0.1-SNAPSHOT.jar to
/app/tmp/cache/.m2/repository/newbudget/NewBudgetCICD/0.0.1-SNAPSHOT/NewBudgetCICD-0.0.1-SNAPSHOT.jar
remote:        [INFO] Installing /tmp/build_3b5e2f1d930ff889bf28d171ee518e0a/pom.xml to /app/tmp/cache/.m2/repository/ne
wbudget/NewBudgetCICD/0.0.1-SNAPSHOT/NewBudgetCICD-0.0.1-SNAPSHOT.pom
remote:        [INFO] ------------------------------------------------------------------------
remote:        [INFO] BUILD SUCCESS
remote:        [INFO] ------------------------------------------------------------------------
remote:        [INFO] Total time: 7.396 s
remote:        [INFO] Finished at: 2018-10-24T13:32:10Z
remote:        [INFO] ------------------------------------------------------------------------
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote:        Done: 78.9M
remote: -----> Launching...
remote:        Released v12
remote:        https://newbudgetcicd.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/newbudgetcicd.git
   589f1eb..d840bcb  master -> master
PS C:\newbudgetcicd> heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on                  ... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-regular-59479 as HEROKU_POSTGRESQL_MAUVE_URL
Use heroku addons:docs heroku-postgresql to view documentation
PS C:\newbudgetcicd>
```

Fonte: O Autor.

Ao finalizar o deploy manual a aplicação ficou disponível no link https://newbudgetcicd.herokuapp.com/ como podemos ver na figura 17.

Figura 17 - NewBudget Implantado Heroku



Fonte: O Autor.

4.3.4. Implantação Contínua - Deploy automatizado

Neste momento partiu-se para a automação do deploy via ferramenta de integração contínua. Para isso foi necessário a criação de um token heroku através do comando "heroku auth:token" para autenticar o acesso do contêiner usado pela CI na API de deploy do heroku. Foi criado no GitLab duas variáveis secretas, uma com o token gerado pelo heroku e outra com o nome da aplicação criada no heroku. As variáveis receberam os nomes de $APP_NAME e $AUTH_TOKEN respectivamente.

Foi adicionado ao arquivo .gitlab-ci.yml o estágio de deploy para ser executado somente na branch master, e nele aplicados os comandos para o deploy. No estágio deploy fez se necessário a utilização de uma imagem docker do ruby:2.3, pois a imagem maven:3-jdk-8 usada para os estágios anteriores de build e test não era compatível com a ferramenta DPL, responsável por se conectar ao heroku para implantação da aplicação. Podemos observar o código do estágio deploy na figura 18.

Figura 18 - Pipeline Deploy Automatizado

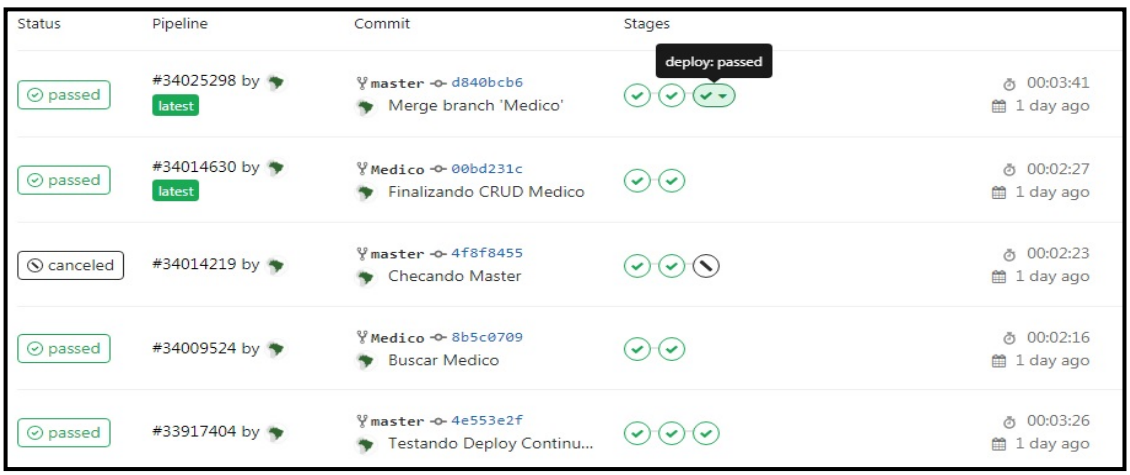
```
18  deploy:
19      image: ruby:2.3
20      stage: deploy
21      script:
22          - apt-get update -qy
23          - apt-get install -y ruby-dev
24          - gem install -q dpl
25          - dpl --provider=heroku --app=$APP_NAME --api-key=$AUTH_TOKEN
26
27      only:
28          - master
```

Fonte: O Autor.

A DPL é uma ferramenta escrita em ruby desenvolvida pela TravisCI, mas que pode ser usada pelo GitLab para a implantação de aplicações em diversos serviços, inclusive o heroku. Para funcionar a ferramenta precisa do Ruby 2.2 ou superior além do Ruby Runtime do ambiente de desenvolvimento Ruby. Para se conectar a API de deploy do heroku é necessário passar o nome da aplicação e o token de autenticação do serviço.

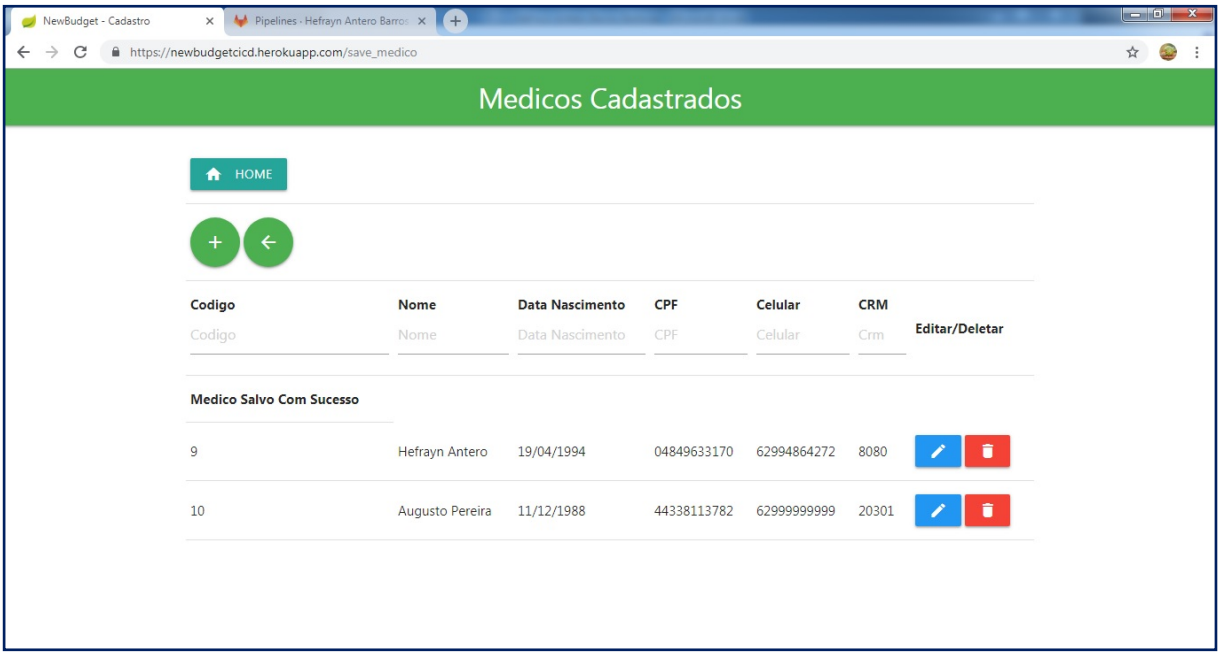
Após implementar o *deploy* automatizado foi realizado o desenvolvimento do CRUD Médicos na *branch* Medico, ao ser realizado o merge na *master* a integração contínua entrou em ação executando o estágio de deploy subindo a funcionalidade automaticamente para o servidor do heroku conforme podemos ver nas figuras 19 e 20.

Figura 19 - Merge Médico/Paciente Deploy Passed



Fonte: O Autor.

Figura 20 - NewBudget Cadastrar Médicos



Fonte: O Autor.

4.3.5. Implantação CD com Kubernets

Ao concluir as implementações com o Heroku foi dado inicio a exploração dos recursos disponíveis pelo Google App Engine. Inicialmente foram criados manualmente três instâncias de máquinas virtuais com sistema operacional Ubuntu 18.10, 3,75 Gb de memória e 10Gb de armazenamento, que serviriam de base para a criação de um cluster Kubernets.

A montagem do cluster requer os seguintes softwares instalados em todos os nós (Máquinas Virtuais): Docker, Kubectl, Kubeadm e Kubelet. O docker é o responsável por executar os contêineres criados pelo kubernets, o kubeadm é o responsável por baixar e configurar o kernel do kubernets na máquina master, bem como conectar os outros nós ao master para criar o cluster, o kubelet responsável por resolver o serviço de nomes entre os nós do cluster, por fim o kubectl é a interface de comunicação entre o administrador do sistema com o kubernets, é através dessa ferramenta que será possível gerenciar o cluster. Com o cluster pronto a saída do comando kubectl get nodes é mostrada na figura 21.

Figura 21 - Kucetl Get Nodes

Fonte: O Autor.

```
root@node1:/home/hefrayn_eu# kubectl get nodes
NAME     STATUS   ROLES    AGE     VERSION
node1    Ready    master   5h11m   v1.12.2
node2    Ready    <none>   5h9m    v1.12.2
node3    Ready    <none>   5h9m    v1.12.2
root@node1:/home/hefrayn_eu#
```

Com o *cluster* funcionando partiu-se para a criação dos pods, que são instâncias de imagens *dockers* que irão compor os serviço a serem executados no cluster. Foram elaborados os arquivos postgres-configmap.yaml e postgres-storage.yaml que tem a função de preparar o ambiente para subir uma imagem do postgres proveniente do docker hub. Nestes arquivos foram incluídos variáveis de ambiente determinando o nome do banco, usuario, senha, bem como configurando um espaço para o contêiner persistir os dados, visto que os contêineres ao serem desligados não persistem dados.

Para subir o contêiner postgres foi elaborado o arquivo postgres-deployment.yaml, neste arquivo é especificada a quantidade de réplicas do contêiner que serão executadas, qual imagem do docker hub será usada, portas que estarão abertas, bem como o nome padrão das réplicas.

Para finalizar é preciso criar uma interface que abstraia o serviço em um local especifico para os consumidores encontrarem o serviço de forma fácil sem a necessidade de conectar-se diretamente às réplicas. Para isso foi criado o arquivo postgres-service.yaml. Com os arquivos prontos basta executá-los na sequência a qual foi mostrada nos parágrafos anteriores com o comando "kubectl create -f <nomedoarquivo.yaml>". Para visualizar o serviço funcionando basta executar o comando "kubect get services"

Com o banco de dados rodando partiu-se para criação de uma imagem docker com o .JAR da aplicação embutido. Para isso foi elaborado o Dockerfile para a elaboração da imagem, nele são passadas instruções de como a imagem deve ser montada. do openjdk 8, na sequência entro da imagem, é passada ndo para executar

Figura 22 - Dockerfile

```
1  FROM openjdk:8
2  COPY target/*.jar /app.jar
3  EXPOSE 8080/tcp
4  ENTRYPOINT ["java", "-jar", "/app.jar"]
5
```

a aplicação. Podemos ver na figura 22 o arquivo Dockerfile.

Fonte: O Autor.

Para construir a imagem roda-se o comando "docker build". Com a imagem pronta é necessário subir essa imagem para o docker hub através do comando "docker push".

Com a imagem pronta para uso é necessário executar os últimos dois passos executados para subir o postgres, o deployment e o service. Neste caso a opção feita foi por não criar os arquivos yaml. No lugar dos arquivos foram utilizadas linhas de comando do kubectl para criar o pod e o serviço da aplicação. Os comando executados foram "kubectl run newbudget --image=nome_da_imagem:versão --port 8080" e " kubectl expose deployment newbudget --type=LoadBalancer".

Neste ponto da implementação o cluster tinha uma replica de cada imagem rodando em contêiner e um serviço para expor cada contêiner como observa-se na figura 23.

Figura 23 - Kubernets Implantado com Falha

```
root@node1:/home/hefrayn_eu# kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
newbudget-656d4c6c69-qsqqm    1/1     Running   0          4h41m
postgres-76555b65bb-7qzf2     1/1     Running   0          5h42m
root@node1:/home/hefrayn_eu# kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)
kubernetes   ClusterIP      10.96.0.1       <none>        443/TCP
newbudget    LoadBalancer   10.97.50.78     <pending>     8080:32675/TCP
postgres     NodePort       10.109.63.216   <none>        5432:32744/TCP
root@node1:/home/hefrayn_eu#
```

Fonte: O Autor.

Como podemos observar, o serviço newbudget apresenta status <pending> na coluna do Ip externo, não sendo possível acessar a aplicação de fora do cluster. Para a resolução do problema seria necessário atribuir o endereço de ip externo de um dos nós ao serviço, bem como liberar e rotear uma porta para o endereço de ip interno do serviço, fato que não ocorreu devido a falta de experiência do acadêmico com redes linux.

Para solucionar o problema tratado no parágrafo anterior e conseguir realizar o *deploy* em um ambiente kubernets, o acadêmico descartou a possibilidade de criar uma cluster manualmente e partiu para o uso da ferramenta Kubernets Engine

disponibilizada pelo Google, pois essa ferramenta abstrai os serviços de redes externas, dispensando esse conhecimento do usuário.

A diferença de um cluster criado manualmente e um criado no Kubernets engine é a abstração dos serviços de rede e a configuração inicial de montagem do cluster, referente a instalação das ferramentas e inicialização dos nós. No Kubernets Engine com um único comando, passando como parâmetro a quantidade de nós, é possível criar um cluster totalmente pronto para uso.

Com a criação do cluster no Kubernets Engine foi refeito exatamente da mesma forma relatada em parágrafos anteriores o deployment do postgres, e o deployment da aplicação. Neste deploy o kubernets aplicou automaticamente um ip externo ao serviço newbudget como podemos observar na figura 24.

Figura 24 - Kubernets Implantado com Sucesso

```
hefrayn_eu@cloudshell:~ (newbudgetcicdg)$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
newbudget-694d5cfc96-gq4j9    1/1       Running   0          2d
postgres-788f6d77b9-4r2m9     1/1       Running   0          2d
hefrayn_eu@cloudshell:~ (newbudgetcicdg)$ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)
kubernetes   ClusterIP      10.39.240.1     <none>          443/TCP
newbudget    LoadBalancer   10.39.241.182   35.198.6.199    8080:31343/TCP
postgres     NodePort       10.39.249.137   <none>          5432:31931/TCP
hefrayn_eu@cloudshell:~ (newbudgetcicdg)$
```

Fonte: O Autor.

Ao acessar pelo navegador o endereço http://35.198.6.199:8080 temos acesso a aplicação funcionando plenamente como podemos constatar na figura 25.

Em caso de alterações na aplicação é necessário remontar uma imagem da aplicação com as alterações e trocar a imagem no pod com o comando "kubectl set image deployment/newbudget newbudget=nome_imagem:versão".

Figura 25 - NewBudget Implantado Kubernets

Fonte: O Autor.

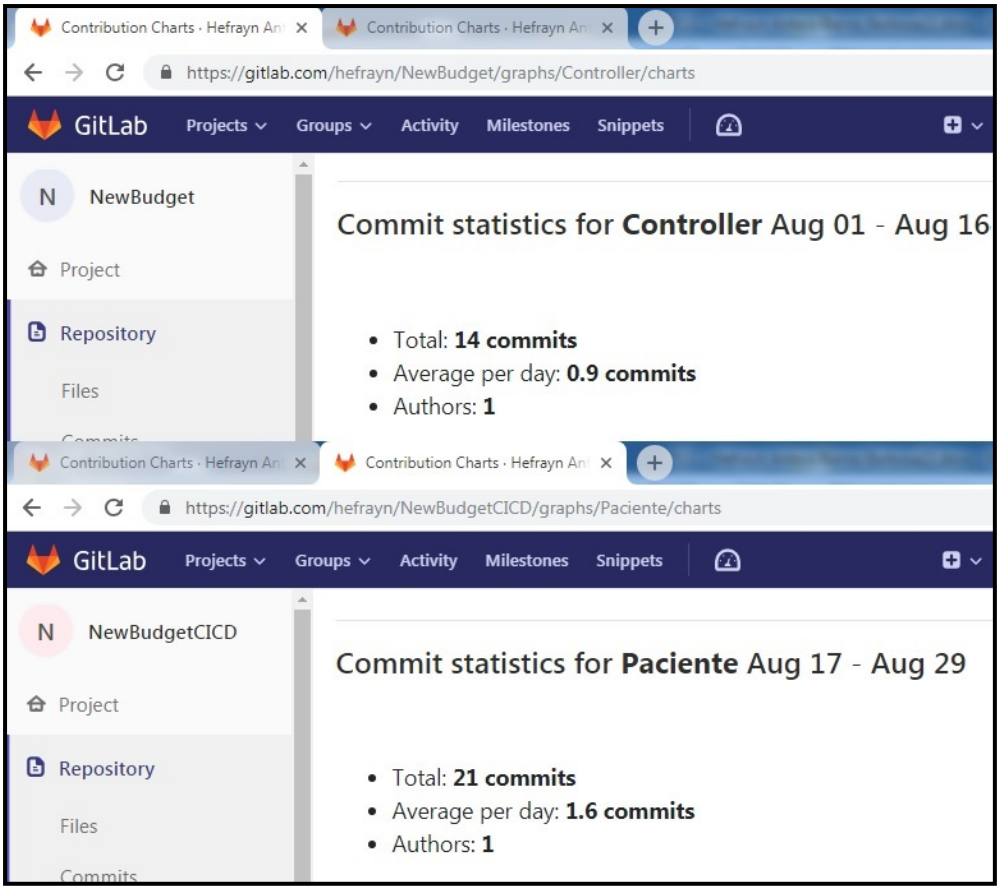
5. CONSIDERAÇÕES FINAIS

O estudo teórico foi fundamental na busca pela implementação dos conceitos de CI/CD, nele foi possível nortear as buscas pelas tecnologias corretas para implementação dos conceitos. Percebeu-se que existem diversas tecnologias disponíveis e a escolha, em grande parte, se dá pela preferência às tecnologias com maior grau de utilização da comunidade.

Ao iniciar as implementações o acadêmico reaproveitou parcialmente o desenvolvimento de um software, adaptando o mesmo às boas práticas exigidas pela integração contínua, aplicando uma metodologia de desenvolvimento alinhada com a prática constante de testes exigida pela CI. No desenvolvimento já era utilizado o sistema de versionamento Git associado ao repositório de código GitLab que, após o estudo teórico, se manteve. No levantamento das tecnologias para repositório e integração contínua percebeu-se que o repositório em uso disponibilizava uma ferramenta de integração contínua integrada, levando a manter a tecnologia que vinha associada a outras vantagens em relação às tecnologias concorrentes.

A implementação da integração contínua foi um sucesso com as ferramentas escolhidas, melhorando o processo de detecção de erros, além de reduzir a ocorrência dos mesmos. A frequência de erros e o tempo gasto corrigindo-os eram maiores na ausência das práticas de integração contínua. Esses fatores contribuíram para o desenvolvedor dispensar mais tempo produzindo. Sem a aplicando os conceitos em um período de 16 dias foram realizados 14 *commits*, contra 21 *commits* realizados em 13 dias como podemos ver na Figura 26.

Figura 26 - Estatísticas Commits



Fonte: O Autor.

Com as práticas implementadas ao ocorrer erros no desenvolvimento, há uma pronta notificação por email pela ferramenta de integração contínua, dando a possibilidade de correção imediata do problema no código, reduzindo consideravelmente o tempo de correção.

A implementação dos conceitos de Entrega Contínua e Implantação Contínua com o Heroku associada com a integração contínua foram realizadas com sucesso, onde a cada commit as funcionalidades eram validadas pela integração contínua e automaticamente eram implantadas no servidor heroku. Porém, é valido ressaltar que foram utilizadas ferramentas gratuitas do heroku. Ao implantar aplicações grátis o Heroku limita a quantidade de dados armazenados, além de hibernar a aplicação na ausência de uso. Quando há uma requisição com a aplicação hibernada o Heroku acorda a aplicação, levando um tempo considerável para a aplicação voltar ao seu estado normal.

No Google App Engine foi implementado somente o conceito de entrega contínua, onde após feedback positivo da integração o deploy deve ser realizado manualmente. Foi criado um cluster Kubernets a partir de 3 instâncias de VMs e posteriormente utilizada o Kubernets Engine para implantar a aplicação. Vale ressaltar que a tecnologia Kubernets é bastante robusta e mais complexa do que o Heroku, dando suporte a implantação de aplicações de qualquer linguagem, porém, requer um alto grau de compreensão de sua estrutura e do sistema operacional utilizado para compor os nós.

No estudo da ferramenta foi possível compreender os principais conceitos da arquitetura do Kubernets, porém houve dificuldades em manejar a saída da aplicação para fora do cluster, problema resolvido com a utilização da plataforma Kubernets disponibilizada pelo Google. O Kubernets Engine abstrai a construção do cluster bem como os serviços de rede externa. Nesta tecnologia foi implementado o contínuous delivery com sucesso.

Com o custo um pouco mais elevado em relação as demais tecnologias, o kubernets associado ao docker em ambientes cloud (nuvem) é o que há de mais avançado no universo das ferramentas disponíveis para a implementação de CI/CD. Na análise da ferramenta o autor utilizou aproximadamente U$ 124,00 dos U$ 300,00 dólares disponibilizados pela Google para a avaliação gratuita da plataforma.O período de análise do Kubernets na plataforma foi de 24 dias, onde o custo diário foi de aproximadamente R$ 18,00. Em casos de projetos com aporte financeiro inicial baixo o autor não recomenda o uso da plataforma.

7. REFERÊNCIAS BIBLIOGRÁFICAS

ANONYMOUSHACKER. **Top 5 Repositórios de Código Fonte Grátis**. Disponível em: <https://www.anonymoushacker.com.br/2018/09/top-5-repositorios-de-codigo-fonte.html>. Acesso em 02 set. 2018.

ALMEIDA, G. A. M. **Fatores de escolha entre metodologias de software tradicionais e ágeis.** São Paulo: Escola Politécnica, Universidade de São Paulo, 2017. Disponível em <http://www.teses.usp.br/teses/disponiveis/3/3136/tde-11042017-143311/publico/GuilhermeAugustoMachadodeAlmeidaCorr17.pdf>. Acesso 14 set. 2018.

APACHE. **What is Maven**. Disponivel em: < https://maven.apache.org/what-is-maven.html>. Acesso em 19 ago. 2018.

ARAÚJO, M. A. P., TRAVASSOS, G. H. **Em busca de um Framework para estudos experimentais em evolução de software**. Vila Velha: WORKSHOP DE MANUTENÇÃO DE SOFTWARE MODERNA, 2006.

AZERI, I. **What Is CI/CD.** 2017. Disponível em: < https://dzone.com/articles/what-is-cicd> . Acesso em: 19 abr. 2018.

BARBOSA, E.; MALDONADO, J.C.; VINCENZI, A.M.R.; DELAMARO, M.E; SOUZA, S.R.S. e JINO, M.. **Introdução ao Teste de Software.** São Paulo: XIV Simpósio Brasileiro de Engenharia de Software, 2000.

BASSI F. D. L. **Experiências com desenvolvimento ágil**. São Paulo: Dissertação (Mestrado)- Universidade de São Paulo, 2008.

BECK, K. **Extreme Programming Explained: Embrace change**. Boston: Addison-Wesley, 1999.

BECK, K. **eXtreme Programming Explained**. Boston: Addison Wesley, 2000.

BECK, K. **Programação Extrema (XP) Explicada: Acolha as Mudanças**, Porto Alegre: Bookman, 2004.

BORGES, E. N. **Conceitos e Benefícios do Test Driven Development**. Porto Alegre: Universidade Federal do Rio Grande do Sul / Instituto de Informática, 2006.

COMAN, H. **Eclipse, NetBeans or IntelliJ: Which is the best Java IDE?**. Jaxenter. [S.I.]. 2014. Disponivel em: <https://jaxenter.com/eclipse-netbeans-or-intellij-which-is-the-best-java-ide-107980.html#authors-block>. Acesso em 07 ago. 2018.

DB-ENGINES. **DB-Engines**. 2018. Disponível em: <https://db-engines.com/en/> Acesso em 05 set. 2018.

DEVMEDIA. **Sistemas de Controle de Versão em Projetos Web**. 2014 Disponível em: <https://www.devmedia.com.br/sistema-de-controle-de-versao-em-projetos-web/31876> Acesso em 28 jul. 2018.

DIAS, A. F. **Conceitos básicos de controle de versão de software - Centralizado e Distribuido**. Pronus Engenharia de Software. [S.I.]. 2016. Disponível em: < https://blog.pronus.io/posts/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/>. Acesso em 18 set. 2018.

DIONISIO, E. J. **PostgreSQL Tutorial**. Devmedia. [S.I.]. 2015. Disponível em: <https://www.devmedia.com.br/postgresql-tutorial/33025> Acesso em 19 ago. 2018.

DUVALL, P.; MATYAS, S.; GLOVER, A. **Contínuous Integration: Improving Software Quality and Reducing Risk.** [S.I.]: Addison-Wesley, 2007. (A Martin Fowler signature book).

ELLINGWOOD, J. **Uma Introdução ao Kubernetes**. DigitalOcean. [S.I.]. 2018. Disponível em: <https://www.digitalocean.com/community/tutorials/uma-introducao-ao-kubernetes-pt>. Acesso em 04 set. 2018.

FARLEY, D e HUMBLE, J. **Entrega Contínua: Como entregar software de forma rápida e confiáve**l. Porto Alegre: Bookman, 2014.

FEITOSA, D. S. **Um estudo sobre o impacto do uso de desenvolvimento orientado por testes na melhoria da qualidade de software**. Universidade Federal da Bahia - Instituto de Matemática, 2007.

FERREIRA, A. R.. **UTILIZAÇÃO DA METODOLOGIA TDD PARA DESENVOLVIMENTO DE SOFTWARE**. Centro Universitário Eurípides de Marília – Univem. 2011 Disponivel em: <http://aberto.univem.edu.br/bitstream/handle/11077/358/Utiliza%C3%A7%C3%A3o%20da%20Metodologia%20TDD%20para%20Desenvolvimento%20de%20Software.pdf?sequence=1>. Acesso em 19 fev. 2018.

FOWLER, M. **Contínuous IntegrationMaio.** 2006. Disponível em: <http://martinfowler.com/articles/contínuousIntegration.html> . Acesso em 11 ago. 2018.

GAMA, A. Test **Driven Development: TDD Simples e Prático**. Devmédia. [S.I.]. 2010. Disponível em: <https://www.devmedia.com.br/test-driven-development-tdd-simples-e-pratico/18533>. Acesso em 24 set. 2018.

GENTIL, E. **Introdução ao Spring Framework**. Devmedia. [S.I.]. 2012 Disponível em: <https://www.devmedia.com.br/introducao-ao-spring-framework/26212> Acesso em: 09 ago. 2018.

GIL, A. C. **Métodos e técnicas de pesquisa social**. São Paulo: Atlas, 5. Ed. 2007.

GIT. **Primeiros Passos - Sobre Controle de Versão**. Disponivel em: <https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Vers%C3%A3o>. Acesso em 15 ago. 2018

GONÇALVES JR., J. P. **O uso da metodologia XP no desenvolvimento de software e os impactos na gestão de riscos**. Monografia (Sistemas de Informação) - Centro Universitário da Fundação Octávio Bastos, São João da Boa Vista, 2009.

LACERDA, E. **Sistemas de Controle de Versão** . Devmedia. [S.I.]. 2012. Disponivel em: < https://www.devmedia.com.br/sistemas-de-controle-de-versao/24574>. Acesso em 25 nov 2018

LIMA, A, T. **Testes De Integração Contínua Com Foco Na Qualidade: Um**

Estudo De Caso Utilizando Jenkins, Maven E Git No Contexto De

Desenvolvimento Dos Projetos Eada E Sgv. Universidade Estadual do Paraná. 2011. Disponível em:<http://www.uece.br/computacaoead/index.php/downloads/doc_download/2150-tccadoniastorreslima >. Acesso em 10 ago. 2018.

MEDEIROS, M. **JUnit Tutorial**. Devmedia. [S.I.]. 2009. Disponível em: < https://www.devmedia.com.br/junit-tutorial/1432>. Acesso em 22 abr 2018.

MEDEIROS, H. **Introdução ao Extreme Programming**. Devmedia. [S.I.]. 2013. Disponível em: < https://www.devmedia.com.br/introducao-ao-extreme-programming-xp/29249>. Acesso em 19 dez 2017.

MEDEIROS, H. **Principais ferramentas de apoio ao desenvolvimento**. Devmedia. [S.I.]. 2016. Disponível em: < https://www.devmedia.com.br/principais-ferramentas-de-apoio-ao-desenvolvimento-java/34126>. Acesso em 27 set 2018.

MEDEIROS, H. **Introdução ao MongoDB**. Devmedia. [S.I.]. 2014. Disponível em: <https://www.devmedia.com.br/introducao-ao-mongodb/30792>. Acesso em: 16 ago. 2018.

MEDEIROS, M. **Extreme Programming - Conceitos e Práticas**. Devmedia. [S.I.]. 2013. Disponível em: < https://www.devmedia.com.br/extreme-programming-conceitos-e-práticas/1498>. Acesso em 17 set 2018.

MENDES, P. **Integração Contínua, Por Martin Fowler**. Pedro Mendes. [S.I.]. 2009. Disponível em: < http://www.pedromendes.com.br/2009/03/04/traducao-do-artigo-sobre-integracao-contínua/> . Acesso em 13 ago. 2018.

MILANI, A. **PostrgreSQL – Guia do Programador**. São Paulo: Novatec, 2008.

PÁDUA, F. W. **Engenharia de Software: Fundamentos, Métodos e Padrões**. Rio de Janeiro: LTC-Livros Técnicos e Científicos Editora S.A. 2ª. ed. 2003.

PAGOTTO, T. Virtualização de Containers (Docker) X VMs. 2017. Disponível em: <https://www.aprendacloud.com.br/containers-docker-vms/>. acesso em: 26 set. 2018.

PEDRYCZ, Witold; PETERS, J. F. **Engenharia De Software**. 3ª. ed. Rio de Janeiro: Elsevier, 2001.

PERRY, J. S. **Informações básicas sobre o Spring Boot**. IBM. [S.I.]. 2017. Disponível em: <https://www.ibm.com/developerworks/br/library/j-spring-boot-basics-perry/index.html> Acesso em: 08 ago. 2018.

PRESSMAN, R. **Engenharia de Software: Uma abordagem Profissional**. Porto Alegre : Sétima Edição.AMGH. 2011.

ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C. et al., "**Qualidade de software – Teoria e prática**". São Paulo: Prentice Hall. 2001.

ROCHA, F. **Integração contínua uma introdução ao assunto**. Devmedia. [S.I.]. 2013. Disponível em: <https://www.devmedia.com.br/integracao-contínua-uma-introducao-ao-assunto/28002> Acesso em: 04 ago. 2018.

ROCHA, F. **Introdução ao desenvolvimento guiado por testes tdd com junit**. Devmedia. [S.I.]. 2012 Disponível em: <https://www.devmedia.com.br/introducao-ao-desenvolvimento-guiado-por-teste-tdd-com-junit/26559>. Acesso em 04 ago. 2018.

RAMOS, J. **Introdução ao Docker**. 2016. Disponível em: <https https://blog.taller.net.br/introducao-ao-docker/>. acesso em: 26 set. 2018.

SANTANA, E. **Desenvolvendo uma aplicação Web com Spring Boot e Spring MVC**. Devmedia. [S.I.]. 2016. Disponivel em: <https://www.devmedia.com.br/desenvolvendo-uma-aplicacao-web-com-spring-boot-e-spring-mvc/34122>. Acesso em 10 ago. 2018.

SANTOS, A. **MySQL: Quem é você?**. Devmedia. [S.I.]. 2006. Disponível em: <https://www.devmedia.com.br/mysql-quem-e-voce/1752>. Acesso em 05 ago. 2018.

SCUDERO, E. **Os 6 frameworks front-end mais amados no mundo (segundo o GitHub)**. Becode. [S.I.]. 2018. Disponível em: <https://becode.com.br/frameworks-front-end-mais-amados-segundo-github/>. Acesso em: 12 ago. 2018.

SONATYPE. **5 Beneficios essenciais de um gerenciador de repositórios**. 2018 Disponivel em: < http://instruct.com.br/5-beneficios-essenciais-de-um-gerenciador-de-repositorios/> Acesso em 01 set. 2018.

SOMMERVILLE, I. **Engenharia de software**. São Paulo: Addison-Wesley, 6. ed. 2003.

SPRING.IO. **Guides**. 2018. Disponivel em: <https://spring.io/>. Acesso em 15 ago. 2018.

STACKSHARE. **Stackshare**. 2018. Disponível em: <https://stackshare.io/>. Acesso em 06 set. 2018.

STORI, D. **Primeiros passos com PaaS com Heroku**. Devmedia. [S.I.]. 2013. Disponível em: <https://www.devmedia.com.br/primeiros-passos-em-paas-com-heroku/29465> Acesso em 05 ago. 2018.

TEIXEIRA, J. R. **Introdução ao MySQL.** Devmedia. [S.I.]. 2013. Disponível em: <https://www.devmedia.com.br/introducao-ao-mysql/27799>. Acesso em: 14 ago. 2018.

TELES, V. M. **Extreme Programming: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade**. São Paulo: Novatec, 2004.

TRUCCO, C. Docker **Compose: O que será? Para que serve? O que come?**. Concrete. [S.I.]. 2017. Disponível em: <https://www.concrete.com.br/2017/12/11/docker-compose-o-que-e-para-que-serve-o-que-come/> Acesso em 17 ago. 2018.

UCHÔA, J. P. **Evolução da metodologia do desenvolvimento de sistemas**. linhadecódigo. [S.I.]. 2008. Disponível em <http://www.linhadecodigo.com.br/artigo/2108/evolucao-da-metodologia-do-desenvolvimento-de-sistemas.aspx>. Acesso em 14 set. 2018.

VASCONCELOS, F. M. **Porque o Kubernets ainda é tão popular com o Docker, mesmo depois do Docker Swarm?**. iMaster. [S.I.]. 2017. Disponível em: <https://imasters.com.br/cloud/por-que-o-kubernetes-ainda-e-tao-popular-com-o-docker-mesmo-depois-do-docker-swarm>. Acesso em: 03 set. 2018.

VENDRAMINI, G. S.. **Um estudo sobre o TDD: Desenvolvimento guiado por testes**. Universidade estadual de Lonrina 2013. Disponivel em: <http://www.uel.br/cce/dc/wp-content/uploads/VersaoPreliminarTCC-GustavoVendramini.pdf>. Acesso em 09 ago. 2018.

VISWANATHAN, P. **Comparing Java IDEs: Eclipse vs. NetBeans vs. IntelliJ**. LifeFire. [S.I.]. 2018. Disponível em: < https://www.lifewire.com/comparing-java-ides-eclipse-vs-netbeans-vs-intellij-2373152> Acesso em 06 ago. 2018.

WAYNER, P. **GitHub vs Bitbucket vs GitLab: Uma batalha épica pelo mindshare desenvolvedor**. CIO. [S.I.]. 2018. Disponível em: <http://cio.com.br/tecnologia/2018/07/13/github-vs-bitbucket-vs-gitlab-uma-batalha-epica-pelo-mindshare-desenvolvedor/> Acesso em: 03 ago. 2018.

WILLIAMS, A. **CI / CD With KUBERNETES**. The New Stack. 2018. Disponível em: <https://thenewstack.io/ebooks/>. Acesso em 29 ago. 2018.