

CENTRO UNIVERSITÁRIO ALVES FARIA
PROGRAMA DE PÓS-GRADUAÇÃO *LATO SENSU*
ARQUITETURA E ENGENHARIA DE *SOFTWARE*

Alex Ferreira de Almeida

Arthur Caetano Borges Silva

Édipo José de Campos

Felipe Carvalho de Saboya

PIPELINE DE ENTREGA CONTÍNUA COM SOFTWARES LIVRES

GOIÂNIA

SETEMBRO DE 2018

CENTRO UNIVERSITÁRIO ALVES FARIA
PROGRAMA DE PÓS-GRADUAÇÃO *LATO SENSU*
ARQUITETURA E ENGENHARIA DE *SOFTWARE*

Alex Ferreira de Almeida

Arthur Caetano Borges Silva

Édipo José de Campos

Felipe Carvalho de Saboya

PIPELINE DE ENTREGA CONTÍNUA COM SOFTWARES LIVRES

Artigo apresentado ao Programa de Pós-Graduação *Lato Sensu* do Centro Universitário Alves Faria, como pré-requisito para a disciplina de Trabalho de Conclusão da Pós-Graduação.

GOIÂNIA
SETEMBRO DE 2018

CENTRO UNIVERSITÁRIO ALVES FARIA
PROGRAMA DE PÓS-GRADUAÇÃO *LATO SENSU*
ARQUITETURA E ENGENHARIA DE *SOFTWARE*

Alex Ferreira de Almeida

Arthur Caetano Borges Silva

Édipo José de Campos

Felipe Carvalho de Saboya

PIPELINE DE ENTREGA CONTÍNUA COM SOFTWARES LIVRES

Trabalho de Conclusão da Pós-Graduação apresentando para avaliação em
____/____/____, tendo sido considerado () **Aprovado** - () **Reprovado** pela Banca
Examinadora.

MEMBROS DA BANCA EXAMINADORA

Prof^a. Mestre Fabiana Rocha de Andrade e Silva - UniALFA
(Orientadora)

Prof. Mestre Thales Bailero Takáo - UniALFA
(Professor Leitor)

Prof. Especialista Murilo Almeida Pacheco - UniALFA
(Professor Leitor)

GOIÂNIA
SETEMBRO DE 2018

RESUMO

Este artigo tem o objetivo de tratar sobre entrega contínua, um conjunto de conceitos, técnicas e ferramentas que permitem fazer a automação de todo o processo de desenvolvimento de um *software*, contemplando desde o registro de uma mudança até a sua implantação, seja em um ambiente de produção ou homologação. A abordagem será feita através de um estudo teórico de conceitos essenciais para a sua compreensão, acompanhado de uma demonstração prática por meio de um cenário hipotético. Ao final, pretende demonstrar que é possível criar um *pipeline*¹ de entrega contínua usando apenas ferramentas de código aberto e as vantagens provenientes de sua utilização.

Palavras-chave: Integração Contínua. GCS. TDD. DevOps. Inspeção de código automática.

¹ Cadeia de elementos processados sequencialmente, organizados de forma que a saída de cada elemento, é a entrada do próximo.

INTRODUÇÃO

Segundo Humble e Farley (2014), a entrega contínua promove a colaboração entre as equipes e permite que as versões do *software* estejam prontas para qualquer ambiente, a qualquer momento, através da automação dos processos executados durante o ciclo de desenvolvimento.

Inevitavelmente, um sistema, durante o seu ciclo de vida, será submetido a mudanças. Estas podem ser de caráter corretivo, preventivo, adaptativo ou evolutivo, e influenciadas por motivos diversos, tais como: mudanças de regras de negócio, alterações na legislação ou questões intrínsecas ao desenvolvimento (MOLINARI, 2007).

Mediante a utilização de técnicas e ferramentas, a entrega contínua possibilita a implantação dessas mudanças por meio de um alto grau de automação de processos, de forma rápida, segura e frequente. Ou seja, permite entregar *software* com mais qualidade, em menos tempo e, consequentemente, com menor custo (HUMBLE; FARLEY, 2014).

Este tema foi escolhido por se tratar de um assunto de grande relevância, pois pode ser aplicado a qualquer empresa de desenvolvimento de *software*. Apesar disso, na prática, ainda é pouco explorado por uma parcela significativa deste segmento, seja por falta de conhecimento dos gestores e/ou mão-de-obra qualificada.

Nos tópicos a seguir, serão apresentados primeiramente os conceitos que constituem a entrega contínua. Para cada um deles, será exibida uma ferramenta, que será utilizada posteriormente no cenário prático para demonstração de um *pipeline* de entrega contínua. E, por último, serão expressadas as considerações finais.

O resultado esperado com a elaboração deste artigo científico é demonstrar que a entrega contínua contribui de forma significativa com o processo de entrega de *software* de maneira automatizada, segura, rápida e de custo mais baixo. Além do mais, pretende-se mostrar que é totalmente viável construir um *pipeline* utilizando apenas ferramentas livres.

1 ENTREGA CONTÍNUA

A alta competitividade no mercado exige que as ideias sejam desenvolvidas de maneira rápida. Sato (2014, p. 4) diz que, “essa linha de pensamento que tenta diminuir o tempo entre a criação de uma ideia e sua implementação em produção é também conhecida como ‘**Entrega Contínua**’”.

De acordo com Humble e Farley (2014), a entrega contínua consiste em um conjunto de princípios, práticas e técnicas que permitem gerar novas versões de forma simples, frequente e com baixo risco. Além disso, um dos mais importantes princípios do DevOps² está presente na entrega contínua: o envolvimento de maneira colaborativa entre todos os envolvidos no processo de desenvolvimento de *software* para uma entrega rápida e confiável.

A cultura de colaboração entre os times, fortemente empregada na entrega contínua, pode sofrer resistência quando implantada em organizações que fazem uso do paradigma tradicional de desenvolvimento de *software*. Para minimizar atrasos e riscos, provenientes de uma possível resistência, é fundamental investir em treinamentos. Deste modo, o impacto pode ser amenizado e a entrega contínua implantada de forma eficaz e bem-sucedida (HUMBLE; FARLEY, 2014).

Ainda, segundo Humble e Farley (2014), o conjunto de práticas que constituem a entrega contínua, além de incentivar a colaboração entre os membros das equipes, também prega a automatização do processo de desenvolvimento de *software*. Ou seja, reduzindo consideravelmente o tempo empregado neste processo, e eliminando grande parte da tensão do dia de lançamento de uma nova versão. Pois, torna o processo menos suscetível a erros provenientes de falhas humanas.

Segundo Sato (2014), é importante distinguir a entrega contínua da implantação contínua, que são práticas que costumam ser confundidas. A implantação contínua consiste em fazer a implantação imediata de qualquer mudança que não resulte em erros durante o processo de construção. Enquanto que, a entrega contínua é uma prática em que a mudança pode ser implantada a qualquer momento, de acordo com a necessidade do negócio. Apesar desta tênue diferença, ambas dependem da criação de um *pipeline* de entrega, que é o objetivo deste artigo.

² Prática da Engenharia de Software que promove a colaboração entre as equipes de desenvolvimento (Dev) e operações (Ops).

A seguir, na Figura 1, é realizada a representação gráfica das práticas que constituem a Entrega Contínua.

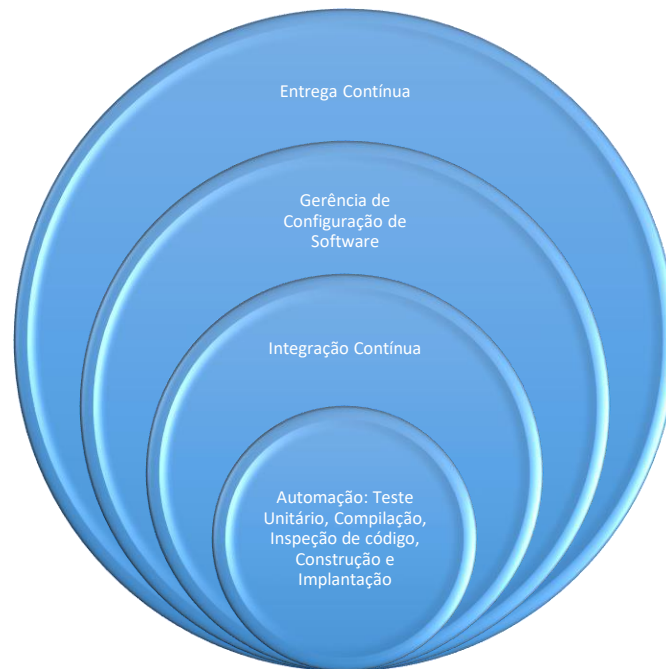


Figura 1: Representação gráfica da entrega contínua
Fonte: Próprio autor

1.1 GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

A Gerência de Configuração de *Software* (GCS) consiste na gestão do ciclo de vida de desenvolvimento de um *software*, de modo a manter um histórico das alterações realizadas, permitindo identificar por quem, quando e por qual motivo foi realizada uma determinada alteração.

A gestão de configuração de software é um conjunto de atividades que foram desenvolvidas para gerenciar alterações através de todo o ciclo de vida de um software. A SCM pode ser vista como atividade de garantia de qualidade de software aplicada através de todo o processo do software (PRESSMAN, 2011, p. 515).

Segundo Pressman (2011), mudanças podem acontecer em qualquer momento. Portanto, a *Software Configuration Management* (SCM - sigla em inglês para GCS) tem atividades que foram desenvolvidas para identificar a alteração; controlar a alteração; assegurar que a alteração esteja sendo implementada corretamente; e, relatar as alterações a outros interessados.

Durante o processo de desenvolvimento de um sistema, são produzidos vários artefatos, como, por exemplo: documentos, modelos e código fonte. Estes artefatos passam por um processo de identificação, e de acordo com sua importância e necessidade de controle, recebem um nome único e passam a ser denominados itens de configuração (ICS). Por sua vez, a um conjunto de ICS dá-se o nome de configuração de *software*.

A configuração de software é, portanto, definida como uma lista dos ICS que compõem o software e suas respectivas versões. Cada uma dessas versões deve ser armazenada de maneira que possa ser recuperada sempre que determinada configuração de software precise ser reproduzida (WAZLAWICK, 2013, p. 222).

Segundo Wazlawick (2013), para a gerência efetiva de um IC, é necessário definir uma *baseline*, que consiste em uma configuração de *software* estável que servirá como base para implementações posteriores. Poderia, por exemplo, ser a *release* de uma versão do *software*, ou seja, a versão de um sistema entregue aos clientes após ser homologada por testes. Deste momento em diante, partindo do pressuposto que foi aprovada por todos os interessados, pode ser modificada apenas mediante a um processo formal de mudança.

Neste ponto, após compreender o básico sobre a GCS, é interessante conhecer algumas atribuições que lhes são incumbidas. Dentre as principais, destacam-se: o controle de versões e o controle de mudanças (WAZLAWICK, 2013).

1.1.1 Controle de Versões

O controle de versão vai rastrear todos os artefatos do projeto (itens de configuração) e manter o controle sobre o trabalho paralelo de vários desenvolvedores através das seguintes funcionalidades:

- a) Manter e disponibilizar cada versão já produzida para cada item de configuração de software.
- b) Ter mecanismos para disponibilizar diferentes ramos de desenvolvimento de um mesmo item, ou seja, diferentes variantes ou variações de um item poderão ser desenvolvidas em paralelo.
- c) Estabelecer uma política de sincronização de mudanças que evite a perda de trabalho e o retrabalho.
- d) Fornecer um histórico de mudanças para cada item de configuração (WAZLAWICK, 2013, p. 222).

Segundo Humble e Farley (2014), o controle de versões, também conhecido como controle de código, é um mecanismo que permite armazenar o histórico de alteração dos arquivos, propiciando desta maneira ter acesso as suas versões anteriores. Também possibilita identificar o que foi modificado; quando foi modificado; por quem foi modificado; e, por quê foi modificado.

De acordo com Wazlawick (2013), dentre as ferramentas livres para controle de versão, pode-se citar: Git, CVS, SVN e Mercurial. Entretanto, optou-se pelo Git, em virtude da sua arquitetura distribuída, com foco em performance e utilizado em grandes projetos de código aberto.

1.1.1.1 Git

Conforme supracitado, o Git é uma ferramenta utilizada no controle de versões de um arquivo qualquer. Trata-se de “uma ferramenta para registrar alterações feitas em conjunto de arquivos ao longo do tempo, uma tarefa tradicionalmente conhecida como ‘controle de versão’.” (SILVERMAN, 2013, p. 9).

O Git descende da mais recente geração de sistemas de controle de versão, baseado no paradigma distribuído, ou seja, difere de outros sistemas mais antigos, como o CVS e SVN, que seguiam a arquitetura centralizada. A grande vantagem consiste no fato de cada cópia do Git possuir uma cópia completa do repositório, inclusive com histórico, permitindo que a maioria dos comandos seja executada mesmo se a rede estiver indisponível (SILVERMAN, 2013).

1.1.2 Controle de Mudanças

O propósito básico do controle de mudanças é ter o controle total de todo e qualquer pedido (ou requisição) de mudança de um produto e de todas as mudanças implementadas. Para cada item de configuração será necessário saber quem, quando, por que e o que mudou, de modo que se possa ter uma rastreabilidade completa entre cada versão modificada ou criada, da antecessora à predecessora (MOLINARI, 2007, p. 51).

Molinari (2007) diz que mudanças são inevitáveis em um produto, sejam em decorrência dos mais variados eventos, tais como, correção de problemas e novas necessidades dos clientes. Neste contexto, o controle de mudanças deve ser formalmente registrado, de modo que possibilite manter a rastreabilidade entre um item de configuração modificado e a solicitação que deu origem a mudança.

Segundo Medeiros (2014), existem várias ferramentas *open source*³ que podem realizar o gerenciamento de mudanças, tais como: GitHub, Mantis, Redmine, Trac e Bugzilla. Neste caso, a ferramenta escolhida foi o GitHub, por possuir uma interface simples, funcionalidades intuitivas e integrada de forma nativa com o sistema de controle de versões (Git).

³ Termo em inglês que se refere a sistemas de código aberto.

1.1.2.1 GitHub

O GitHub é uma aplicação Web, criada em 2008, que possibilita a hospedagem e o compartilhamento de repositórios Git. Atualmente, hospeda inúmeros projetos de código abertos, dentre os quais, destacam-se: jQuery, Node.js, Ruby On Rails, Jenkins, Spring e JUnit (AQUILES; FERREIRA, 2014).

Por meio do GitHub, também é possível realizar o controle de mudanças utilizando os *Issues*. Este recurso, permite registrar as mudanças, categorizá-las, atribuí-las a um responsável e fechá-las quando forem finalizadas (GITHUB, 2014).

1.2 DESENVOLVIMENTO ORIENTADO A TESTES

O *Test Driven Development* (TDD - ou Desenvolvimento Orientado a Testes, em português) é uma técnica que consiste na inversão do paradigma de desenvolvimento de *software* tradicional. Ou seja, os testes unitários são escritos pelos desenvolvedores antes da codificação propriamente dita.

Para aqueles que não estão familiarizados com o conceito, a ideia é que quando uma nova funcionalidade está sendo desenvolvida, ou quando um defeito está sendo corrigido, a primeira atitude que os desenvolvedores devem tomar é a criação de uma especificação executável de qual deve ser o comportamento do código a ser escrito. Esses testes não somente guiam a arquitetura da aplicação, como também servem como testes de regressão e documentação sobre o código e o comportamento esperado da aplicação (HUMBLE; FARLEY, 2014, p.71).

Para Aniche (2012), o TDD propõe que se escreva um teste de unidade, ou teste unitário, antes de escrever o código. Como o próprio nome já sugere, o teste unitário avalia apenas uma unidade do sistema. Geralmente, corresponde a uma classe, quando o paradigma utilizado é a orientação a objetos.

Essa prática faz o desenvolvedor utilizar os testes unitários como uma espécie de rascunho, para escrever o código definitivo posteriormente. Desta maneira, contribui para evitar possíveis custos de um código mal escrito. Além disso, deixar o teste para o final pode trazer prejuízos, pois dificulta a análise de impacto entre as classes (ANICHE, 2012).

Para criação e execução dos testes unitários, optou-se pela utilização do JUnit, por ser a ferramenta mais difundida na comunidade Java, inclusive estando embarcada nas versões mais recentes de ferramentas *Integrated Development Environment* (IDEs - ou Ambiente de Desenvolvimento Integrado, em português).

1.2.1 JUnit

Conforme site oficial do JUnit (s.a.), trata-se de um *framework* para criação de testes unitários. Segundo Aniche (2012, p. 11), é “o framework de testes de unidade mais popular do mundo Java”.

Permite executar os testes unitários, exibindo quais obtiveram êxito e quais falharam, retornando informações importantes para o diagnóstico do problema, tais como: nome do teste e a linha que apresentou a falha (ANICHE, 2012).

1.3 AUTOMAÇÃO DE *BUILDS*

O *build*, também conhecido como construção, é um processo realizado sobre o código fonte de maneira a gerar os arquivos necessários para a execução do *software*. Segundo Sato (2014, p. 131), “O *build* de um sistema envolve tarefas necessárias para executá-lo, como por exemplo: compilação, *download* e resolução de dependências, vinculação com bibliotecas, empacotamento, cálculo de métricas de qualidade etc.”.

Hüttermann (2012) afirma que, quanto mais o sistema cresce, o que é a tendência, vai se tornando mais complexo e difícil recordar de todos os passos necessários para execução do *build*. E que, portanto, a automatização tende a deixar o trabalho mais simplificado, rápido e seguro, já que as linguagens de programação mais utilizadas possuem ferramentas para facilitar este processo.

Algumas ferramentas para automação de *builds* são: Maven, Ant e Make. Para esse item, a ferramenta escolhida foi o Maven. Assim como o JUnit, já vem com suporte nativo nas versões mais recentes dos IDEs mais populares. É uma das ferramentas mais disseminadas entre os desenvolvedores Java, pois além da automação de *builds*, também permite automatizar o gerenciamento das bibliotecas utilizadas nos projetos.

1.3.1 Maven

Segundo Humble e Farley (2014), o Maven é uma ferramenta que permite automatizar o processo de compilação, execução dos testes unitários, construção e implantação, utilizando uma estrutura padrão de diretórios e comandos pré-definidos. Além disso, possibilita o

gerenciamento automático de bibliotecas Java e dependências entre projetos, fazendo uso apenas de algumas linhas da linguagem *eXtensible Markup Language* (XML).

De acordo com o site oficial do Maven (s.a.), trata-se de um acumulador de conhecimento, que permite a reutilização de Java *ARquive* (JARs⁴) entre projetos e pode gerenciar a construção de um projeto mediante ao *Project Object Model* (POM - Projeto de Modelo de Objeto, em português), facilitando a gestão de qualquer projeto baseado em Java.

O Maven possui um ciclo de vida padrão dividido em fases. Dentre as mais importantes, destacam-se: *validate*, *compile*, *test*, *package*, *integration-test*, *verify*, *install* e *deploy*. Estas fases podem ser chamadas de forma individual, via linha de comando, mas durante a execução mantém uma relação hierárquica com as predecessoras. Ou seja, ao executar determinada fase, todas as anteriores serão executadas também (MAVEN, s.a.).

1.4 INTEGRAÇÃO CONTÍNUA

Segundo Humble e Farley (2014), a integração contínua é uma prática originada na metodologia *Extreme Programming* (XP), descrita inicialmente por Kent Beck. O termo integração contínua refere-se a prática de compartilhar alterações locais com os demais integrantes do time constantemente. Tem o intuito de integrar de forma frequente os fontes locais, que estão em processo de desenvolvimento, com os fontes mais recentes do repositório remoto. Ou seja, garantindo que não exista uma grande defasagem entre eles e que o código permaneça estável. Conforme supracitado, depende de alguns pré-requisitos, tais como: controle de versões, testes unitários e *builds* automatizados. No caso específico deste artigo, as ferramentas escolhidas para contemplar estes itens foram, respectivamente: Git, JUnit e Maven.

Integração Contínua exige que a aplicação seja compilada novamente a cada mudança feita e que um conjunto abrangente de testes automatizados seja executado. É fundamental que, se o processo de compilação falhar, o time de desenvolvimento interrompa o que está fazendo e conserte o problema imediatamente. O objetivo da integração é manter o software em um estado funcional o tempo todo (HUMBLE e FARLEY, 2014, p.55).

Os principais objetivos da integração contínua são: entregar *software* de forma rápida e com mais qualidade, evitar conflitos (divergências) provenientes de fontes desatualizados e diagnosticar defeitos mais cedo. Este último item, inclusive, é de suma importância, pois quanto antes os defeitos forem detectados, menor é a quantidade de custo e tempo despendidos para

⁴ Extensão que contém arquivos compactados associados à linguagem Java, tais como: aplicações ou bibliotecas.

corrigi-los. Outro importante aspecto, se refere ao fato do aumento da satisfação do cliente, visto que as versões são entregues de forma mais ágil e segura (HUMBLE; FARLEY, 2014).

De acordo com Sato (2014), para garantir o sucesso da integração contínua, é aconselhável orientar que os desenvolvedores realizem ao mínimo um *commit*⁵ por dia. Para evitar que as alterações fiquem acumuladas localmente e indisponíveis para os demais integrantes da equipe. Além disso, é recomendável configurar um servidor de integração contínua, responsável por disparar a compilação e execução dos testes unitários sempre que um novo *commit* for identificado.

O servidor de integração contínua é responsável por monitorar o repositório central de código e iniciar um *build* do projeto toda vez que detectar um novo *commit*. Ao final da execução de um *build*, ele pode passar ou falhar. Quando o *build* falha, é comum dizer que ele quebrou ou que ele “está vermelho”. Quando o *build* termina com sucesso, é comum dizer que o *build* passou ou que “está verde” (SATO, 2014, p.140).

Portanto, o servidor de integração contínua, também é responsável por divulgar o resultado dos *builds* e manter todo o time de desenvolvimento informado sobre a situação do projeto, seja através de painéis informativos, aplicações *web*, *e-mails* e/ou notificações. É importante ressaltar que, nos casos de quebra, é recomendável que o time priorize a correção e interrompa os *commits* até que a alteração com defeito seja identificada e consertada (SATO, 2014).

Existem várias ferramentas que podem ser utilizadas para configurar um servidor de integração contínua, dentre as quais pode-se citar, por exemplo: Jenkins, CruiseControl e TeamCity (SATO, 2014). Neste caso, optou-se pelo Jenkins, por se tratar de uma ferramenta mantida por uma comunidade bastante ativa, e que propicia a integração simplificada com diversas ferramentas, mediante a grande quantidade de *plugins*⁶ disponíveis.

1.4.1 Jenkins

Segundo o site oficial do Jenkins (s.a.), trata-se de um servidor de integração contínua de código aberto desenvolvido em Java, originado a partir do Hudson⁷ e que conta atualmente com mais de 1000 *plugins*. De acordo com Sato (2014, p. 141), “é uma ferramenta de código

⁵ Confirmar alterações que serão submetidas ao repositório remoto.

⁶ *Softwares* complementares que são acoplados a um programa principal com o intuito de adicionar recursos a ele.

⁷ Ferramenta de integração contínua mantida pela empresa Oracle e que serviu como base para a criação do Jenkins.

livre bem popular no mundo Java, possui um amplo ecossistema de *plugins* e uma comunidade bem ativa”.

1.5 AUTOMAÇÃO DE *DEPLOYS*

No contexto da entrega contínua, *deploy* é um termo utilizado para designar implantação. Segundo Sato (2014), é um processo que consiste em implantar os artefatos, gerados na etapa de *build*, em um ambiente de produção. Tendo como objetivo principal disponibilizar o sistema para utilização por parte dos usuários.

De acordo com Humble e Farley (2014), a automação do *deploy* depende da criação de um *pipeline* de implantação, responsável por fazer a transição do controle de versões até os usuários. Sendo a última etapa do ciclo de entrega, tem o intuito de alcançar um nível que permita fazer a implantação da forma mais simples possível.

Os principais objetivos do *pipeline* de implantação são: reduzir o tempo gasto para disponibilizar um *software* em produção; promover a colaboração e agilizar o *feedback* entre os times (desenvolvimento, operações e testes); e, reduzir os riscos de possíveis problemas nas versões, uma vez que o processo é executado e testado frequentemente (HUMBLE; FARLEY, 2014).

Há diversas formas para automatizar o *deploy* das aplicações, desde a adoção de ferramentas como o Maven e Jenkins, até mesmo a criação de *scripts*⁸. Entretanto, optou-se por utilizar o próprio Jenkins, visto que já está sendo utilizado no projeto e dispõe de um *plugin* destinado a esta finalidade denominado *Deploy Plugin*.

1.6 INSPEÇÃO/ANÁLISE AUTOMATIZADA DE CÓDIGO

Segundo Sampaio (2014), um *software* com baixa qualidade, diminui a produtividade e aumenta os custos da empresa. Diante deste cenário, a análise de código é importante para contribuir com a qualidade do produto, e, consequentemente, aumentar a lucratividade.

Através da análise de código automatizada, é possível detectar possíveis pontos de falhas e vulnerabilidades, bem como, sugestões para corrigi-las. Além disso, é possível gerar métricas, que servem como subsídios para determinar o nível de qualidade de um *software* (SAMPAIO, 2014).

⁸ Conjunto de instruções para executar ações em determinado aplicativo.

Dentre os principais problemas originados por um sistema com baixa qualidade, pode-se citar: prazo maior para realizar manutenções, testes não confiáveis, retrabalho, alto custo, incerteza nos prazos. Com base nos relatórios de análise de código, é importante estabelecer metas de qualidade mínimas para cada item, de modo que possam ser mantidas em um limite aceitável (SAMPAIO, 2014).

A ferramenta escolhida neste caso, foi o SonarQube, pois possui um amplo conjunto de regras pré-definidas para Java e diversos plugins que facilitam a integração com outras ferramentas.

1.6.1 SonarQube

De acordo com o site oficial do SonarQube (s.a.), é uma ferramenta de *software* livre para realizar inspeções automáticas através da análise estática do código fonte. Possibilita detectar falhas e vulnerabilidades de segurança em mais de 20 linguagens de programação. Dentre as suas principais vantagens, destaca-se a sua facilidade de integração com ferramentas de *build* automatizado, integração contínua e controle de versões.

Segundo Camargo (2015), o SonarQube realiza a análise do código considerando sete aspectos: Arquitetura e *Design*, Duplicidades, Teste unitários, Complexidade, Erros em potencial, Padrões de codificação (e violação) e Comentários (insuficientes ou excessivos). Portanto, de acordo com estes elementos e com as configurações realizadas na ferramenta, são gerados os relatórios com as métricas que permitirão aferir a qualidade do projeto.

2 DevOps

Na abordagem tradicional das metodologias de desenvolvimento de *software*, existe uma divisão clara de responsabilidades entre desenvolvimento e operações. Segundo Sato (2014), com a divisão específica de formas de trabalho para cada área, desenvolvimento e operação, a primeira ideia que se tem é que parece fazer sentido. No entanto, apesar do objetivo final ser o mesmo, visando o cliente, cada área tem seu objetivo distinto. Enquanto a área de desenvolvimento trabalha na evolução do sistema através de mudanças, a operação preocupa-se com o risco que tais mudanças trazem à estabilidade do *software*, portanto, as evitam ao máximo. Ou seja, há um conflito de interesses.

Para consolidar a relação entre as duas áreas, fez-se necessária a criação de processos para administrar responsabilidades e modo de trabalho de cada equipe. O processo mostrou-se eficiente até determinado momento, entretanto, com o crescente risco e complexidade de cada *deploy*⁹, o processo se tornou deveras burocrático, resultando em um problema denominado “a última milha” (SATO, 2014).

A última milha se refere à fase final do processo de desenvolvimento que acontece após o software atender todos os requisitos funcionais mas antes de ser implantado em produção. Ela envolve várias atividades para verificar se o que vai ser entregue é estável ou não, como: testes de integração, testes de sistema, testes de desempenho, testes de segurança, homologação com usuários (também conhecido como User Acceptance Tests ou UAT), testes de usabilidade, ensaios de deploy, migração de dados etc (SATO, 2014, p. 4).

Portanto, era necessário simplificar o *deploy*, tornando o processo mais ágil, e menos suscetível a riscos. Para tanto, alguns processos foram automatizados, entre eles, o *deploy*.

Inspirado no sucesso dos métodos ágeis, um novo movimento surgiu para levar a mesma linha de raciocínio para o próximo nível: o movimento DevOps. Seu objetivo é criar uma cultura de colaboração entre as equipes de desenvolvimento e de operações que permite aumentar o fluxo de trabalho completado - maior frequência de deploys - ao mesmo tempo aumentando a estabilidade e robustez do ambiente de produção (SATO, 2014, p. 6).

Conforme Hüttermann (2012), o termo DevOps representa uma junção das palavras desenvolvimento e operações. A primeira palavra, está relacionada a pessoas da área de desenvolvimento de *software*, como: programadores, testadores e garantia de qualidade. Enquanto a segunda se refere a especialistas ligados à área responsável por colocar o *software* em produção e administrar a infraestrutura de produção, tais como: administradores de sistema, administradores de banco de dados e técnicos de rede.

⁹ Termo em inglês que se refere à implantação.

As práticas descritas pelo DevOps que tornam o processo de entrega de *software* mais ágil são: ênfase no aprendizado por *feedbacks*¹⁰ imediatos da produção para o desenvolvimento e a consequente otimização do tempo de ciclo. Ou seja, essas práticas contribuem para agilizar a entrega e produzir *software* de alta qualidade (HÜTTERMANN, 2012).

Portanto, apesar do DevOps não ser indispensável para a criação de um *pipeline* da entrega contínua, pode-se perceber que implicitamente acaba sendo referenciado durante este processo. Ou seja, a entrega contínua aplica conceitos importantes abordados no DevOps, como por exemplo, a colaboração entre os times e a automação dos processos necessários para implantação do *software*.

¹⁰ Termo em inglês referente a retorno.

3 CENÁRIO DE ENTREGA CONTÍNUA

Após apresentar os conceitos e ferramentas necessários para a elaboração de um *pipeline* da entrega contínua, será realizada uma demonstração mediante a criação de um cenário hipotético, que simula desde o registro de uma mudança até a sua implantação em um ambiente de produção, conforme demonstrado na imagem a seguir (Figura 2).

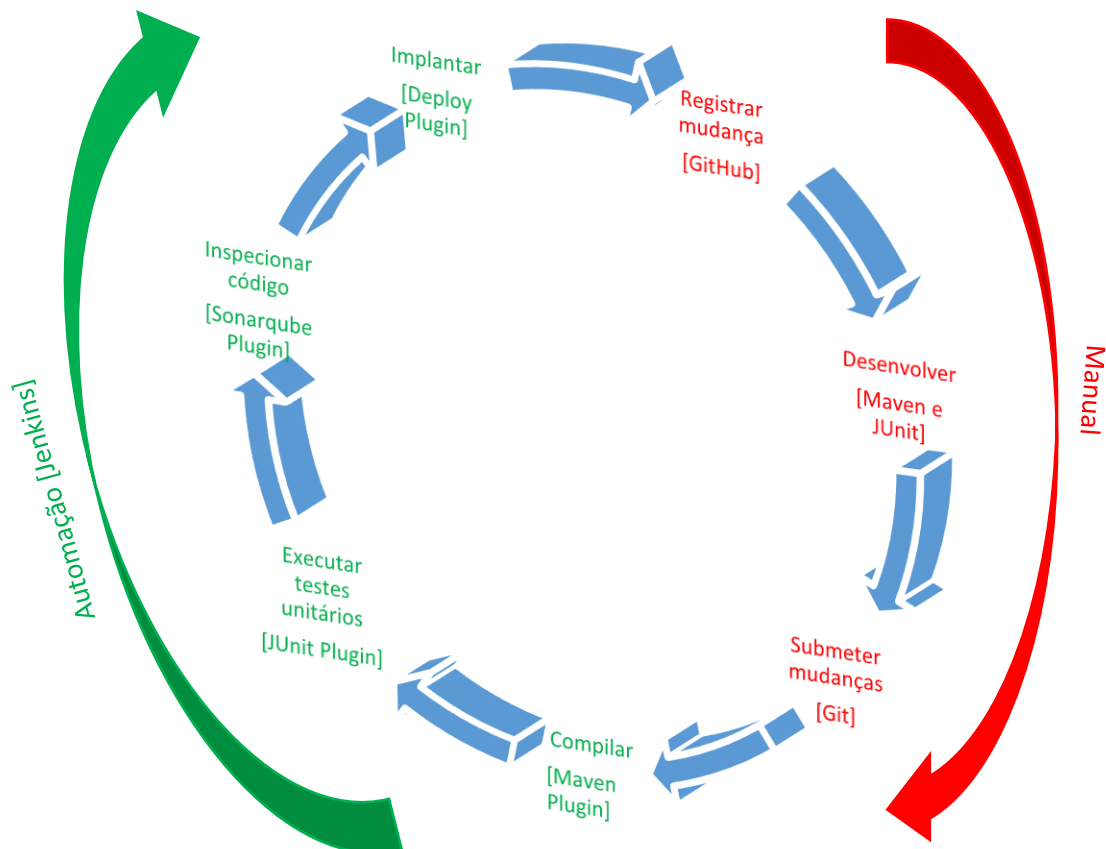


Figura 2: Representação gráfica do cenário de entrega contínua

Fonte: Próprio autor

Foi necessário desenvolver um sistema denominado CalcWeb (vide Figura 3), que consiste em uma calculadora *web* que realiza as quatro operações básicas: adição, subtração, multiplicação e divisão. Para o desenvolvimento, foi utilizada a linguagem de programação Java e algumas bibliotecas, que não serão detalhadas em virtude de serem gerenciadas automaticamente pelo Maven e não fazerem parte do escopo principal do projeto.

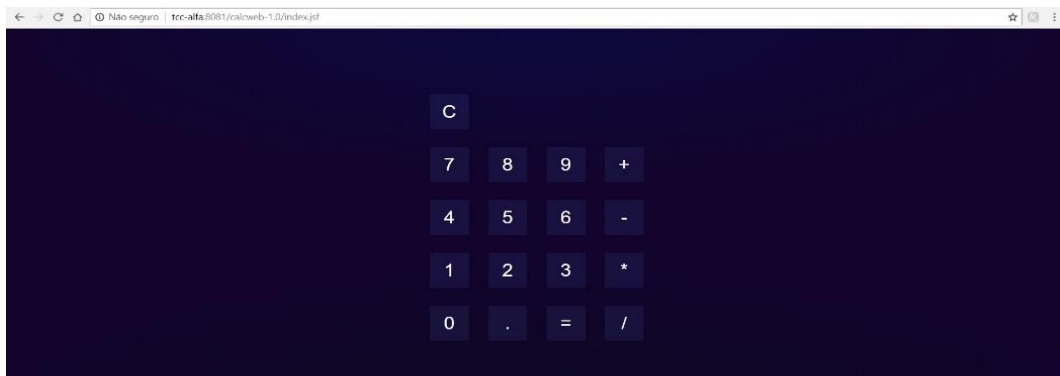


Figura 3: Tela principal do CalcWeb contendo as quatro operações básicas

Fonte: Próprio autor

Foi desenvolvido um *software* relativamente simples, por se tratar de um trabalho de cunho acadêmico, cujo objetivo é demonstrar o ciclo de uma mudança utilizando a entrega contínua. O código fonte está disponível para *download* no GitHub por meio da seguinte URL: <https://github.com/UniALFA-TCC/calweb-ec.git>.

3.1 Pré-requisitos

É importante ressaltar que, além do código fonte citado anteriormente, também foram utilizados os seguintes recursos:

- Servidor de controle de versões e mudanças: GitHub;
- Máquina virtual Linux (Ubuntu);
- Servidor de integração contínua: Jenkins (Imagem Docker), incluindo os *plugins*: Git Plugin, JUnit Plugin, SonarQube Scanner for Jenkins, Maven Project Plugin e Deploy to container Plugin;
- Servidor de aplicação: Apache Tomcat;
- Servidor de inspeção de código: SonarQube (Imagem Docker).

Não serão abordados detalhes destes requisitos, pois também não fazem parte do escopo deste artigo. Além disso, mais detalhes podem ser encontrados facilmente na internet, inclusive em seus respectivos *sites* oficiais.

3.2 Registro da mudança

Para representar a mudança, será realizada a inclusão de mais uma funcionalidade na calculadora, que além das quatro operações básicas, passará a dispor da função de potenciação, representada pelo acento circunflexo (^). O registro da mudança foi realizado pelo GitHub, conforme demonstrado na Figura 4.

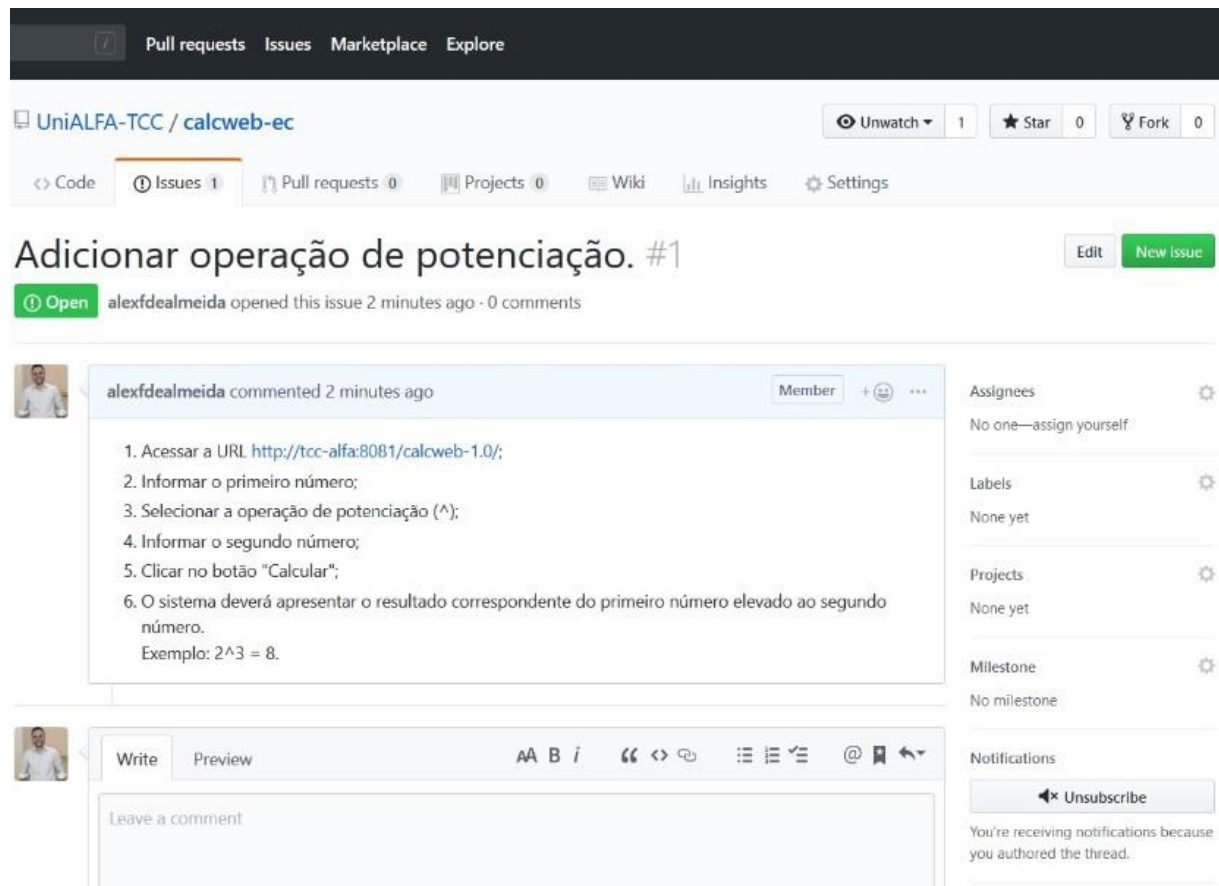


Figura 4: Registro da mudança no GitHub

Fonte: Próprio autor

Conforme pode ser observado, o GitHub oferece várias informações úteis referentes ao registro da mudança, como por exemplo: código (#1), resumo, situação (aberta/fechada), data, relator, comentários e descrição detalhada. Dentre as informações citadas, é importante destacar o código, um identificado único sequencial precedido do caractere “#”, que será informado no *commit* do Git e permitirá fazer o relacionamento entre os sistemas de controle de versão e mudanças.

3.3 Desenvolvimento e submissão das mudanças

Para implementar a nova funcionalidade, é necessário modificar três arquivos: “CalculadoraTest.java”, “CalculadoraMB.java” e “index.xhtml”. No primeiro arquivo (vide Figura 5), é necessário criar o teste unitário para a operação de potenciação. Já no segundo (vide Figura 6), deve-se alterar o método “Calcular”, adicionando uma condição para utilizar o método “pow” da classe “Math”. Por fim, no terceiro arquivo (vide Figura 7), basta adicionar um botão, representado pelo caractere “^”.

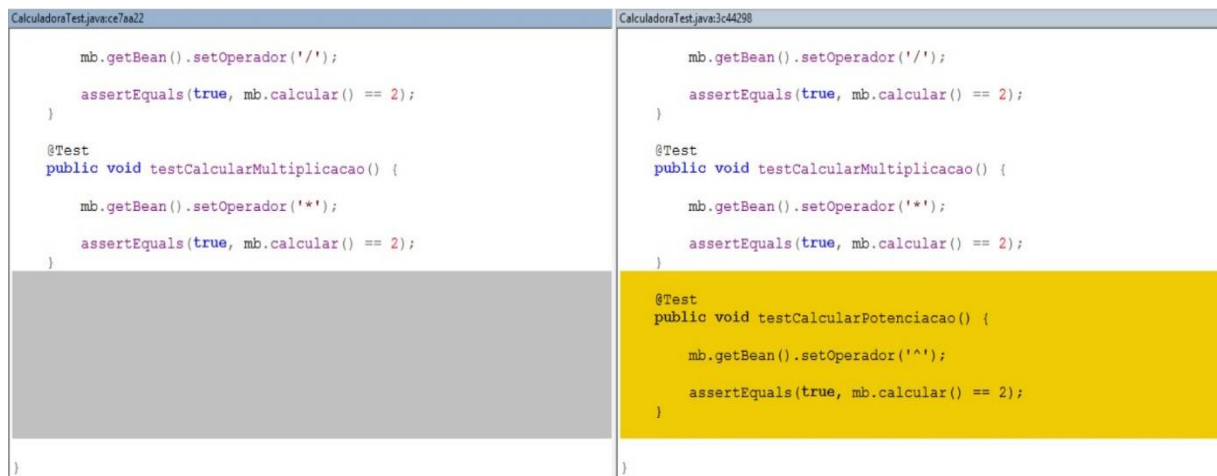


Figura 5: Mudança no arquivo CalculadoraTest.java

Fonte: Próprio autor

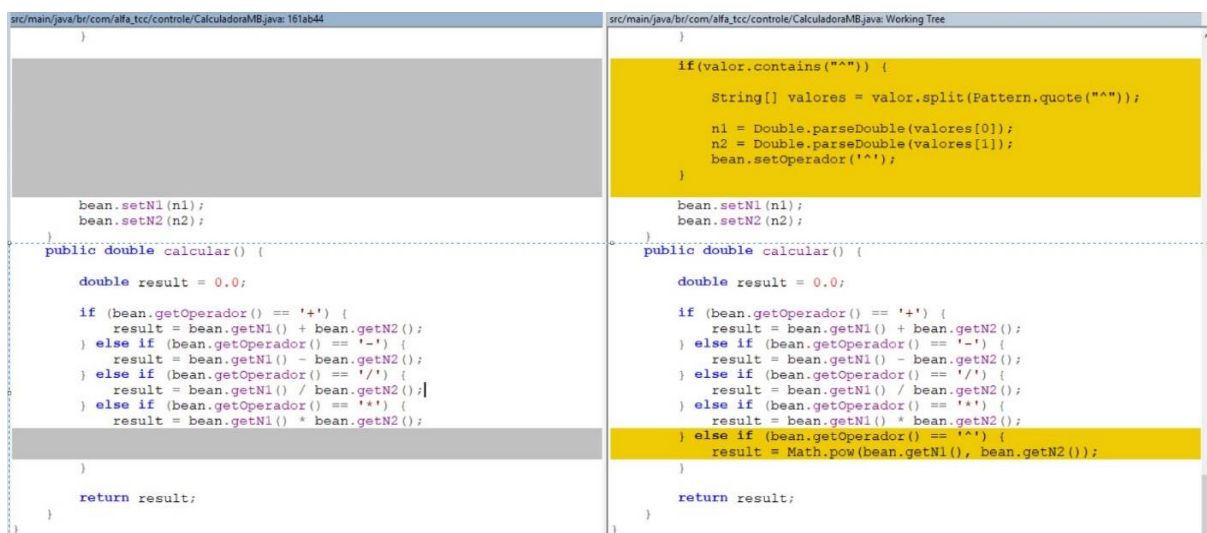


Figura 6: Mudança no arquivo CalculadoraMB.java

Fonte: Próprio autor

src/main/webapp/index.xhtml: 161ab44	src/main/webapp/index.xhtml: e6b0e20
<pre> <button class="num" data-num=".">.</button> <button id="equals" class="equals" data-result="">=</button> <button class="num" data-num=" / ">/</button> </pre>	<pre> <button class="num" data-num=".">.</button> <button id="equals" class="equals" data-result="">=</button> <button class="num" data-num=" / ">/</button> </pre>
<pre> <h:commandButton update="form_calculadora" value="click" action="#{CalculadoraMB.calcularExpressao()}" /> </div> <button id="reset" class="reset">Reset Universe?</button> </h:form> <script> </pre>	<pre> <button class="num" data-num=" ^ ">^</button> <h:commandButton update="form_calculadora" value="click" action="#{CalculadoraMB.calcularExpressao()}" /> </div> <button id="reset" class="reset">Reset Universe?</button> </h:form> <script> </pre>
var operadores = ['+', '-', '*', '/'];	var operadores = ['+', '-', '*', '/', '^'];

Figura 7: Mudança no arquivo index.xhtml

Fonte: Próprio autor

Após realizar as alterações supracitadas, deve-se enviá-las para o servidor do Git. Este processo é feito em duas etapas, executando os seguintes comandos, respectivamente: *commit* (vide Figura 8) e *push* (vide Figura 9). O primeiro comando é executado apenas de forma local, e tem a finalidade de marcar quais mudanças estão aptas a serem enviadas para o servidor. Já o segundo comando, é o que de fato submete os *commits* locais, pendentes, para o repositório remoto do Git.

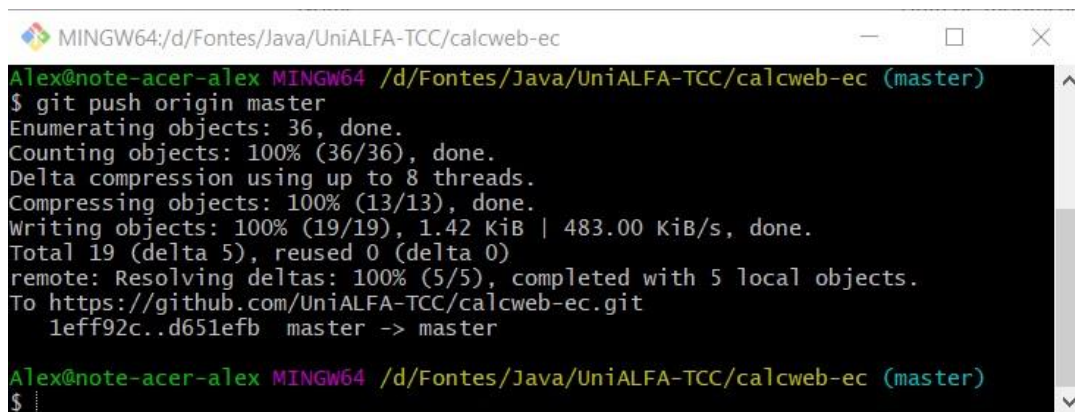
```

MINGW64:/d/Fontes/Java/UniALFA-TCC/calweb-ec
Alex@note-acer-alex MINGW64 /d/Fontes/Java/UniALFA-TCC/calweb-ec (master)
$ git commit -a -m "#1 Adicionada operacao de potenciacao"
[master d651efb] #1 Adicionada operacao de potenciacao
3 files changed, 12 insertions(+)
Alex@note-acer-alex MINGW64 /d/Fontes/Java/UniALFA-TCC/calweb-ec (master)
$

```

Figura 8: Etapa 1: *Commit* das mudanças

Fonte: Próprio autor



```
MINGW64;d/Fontes/Java/UniALFA-TCC/calweb-ec
Alex@note-acer-alex MINGW64 /d/Fontes/Java/UniALFA-TCC/calweb-ec (master)
$ git push origin master
Enumerating objects: 36, done.
Counting objects: 100% (36/36), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (19/19), 1.42 KiB | 483.00 KiB/s, done.
Total 19 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
To https://github.com/UniALFA-TCC/calweb-ec.git
 1eff92c..d651efb master -> master

Alex@note-acer-alex MINGW64 /d/Fontes/Java/UniALFA-TCC/calweb-ec (master)
$
```

Figura 9: Etapa 2: Submissão das mudanças locais para o servidor do Git
Fonte: Próprio autor

Observe que, na mensagem do *commit*, foi feita a referência com a mudança registrada anteriormente no GitHub, através da marcação #1. Deste modo, ao visualizar o histórico de *commits* no Git, é possível identificar com que mudanças se relacionam, permitindo-se criar uma rastreabilidade entre ambas as ferramentas. Para ter acesso a estas informações, basta digitar o comando “gitk”, conforme Figura 10.

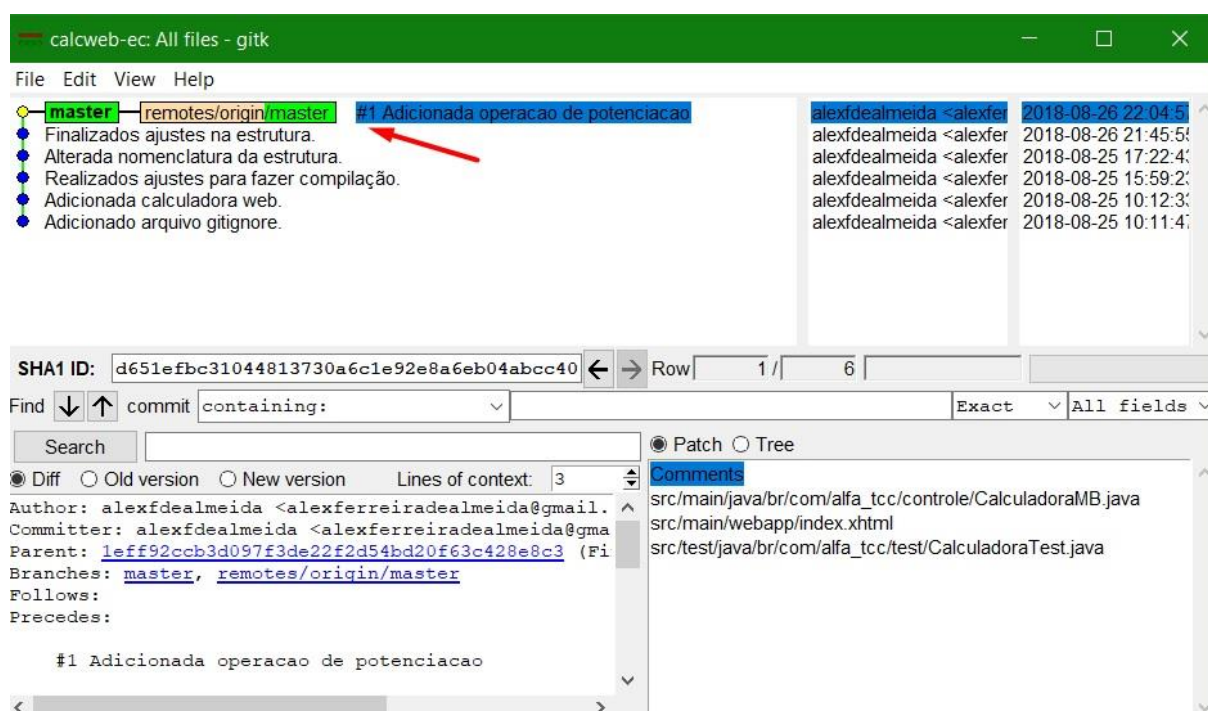


Figura 10: Visualização do histórico de *commits* do Git utilizando o comando gitk
Fonte: Próprio autor

3.4 Compilação, execução dos testes unitários, inspeção de código e implantação

A partir deste ponto, todas as etapas posteriores são realizadas de forma automatizada pelo servidor de integração contínua. De acordo com as configurações pré-definidas, o Jenkins monitora a cada minuto se o servidor Git recebeu novas alterações. Ao identificar um novo *commit*, aciona o *plugin* do Maven para fazer a compilação (vide Figura 11) e execução dos testes unitários (vide Figura 12). Em seguida, aciona o SonarQube Plugin, que realiza a inspeção de código (vide Figura 13) e gera as métricas de qualidade. Por último, executa o *plugin* que faz o *deploy*, responsável por implantar (vide Figura 14) a nova versão em ambiente de produção.

Cada etapa do processo de *build* pode ser acompanhada em tempo real ou acessada posteriormente através da saída de console. O *log* gerado permite visualizar se o processo finalizou com sucesso ou apresentou alguma falha. Sendo que, caso ocorra algum problema, permite realizar a notificação por e-mail aos destinatários previamente informados nas configurações do projeto.



Saída do console

```
Started by an SCM change
Building in workspace /var/jenkins_home/workspace/calweb-ec
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/UniALFA-TCC/calweb-ec.git # timeout=10
Fetching upstream changes from https://github.com/UniALFA-TCC/calweb-ec.git
> git --version # timeout=10
using GIT_ASKPASS to set credentials
> git fetch --tags --progress https://github.com/UniALFA-TCC/calweb-ec.git +refs/heads/*:refs/remotes/origin/*
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision d651efbc31044813730a6c1e92e8a6eb04abcc40 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f d651efbc31044813730a6c1e92e8a6eb04abcc40
Commit message: "#1 Adicionada operacao de potenciacao"
> git rev-list --no-walk 1eff92ccb3d097f3de22f2d54bd20f63c428e8c3 # timeout=10
[calweb-ec] $ /var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/maven-3.5.2/bin/mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building calweb 1.0
-----
```

Figura 11: Identificação de novo *commit* e compilação

Fonte: Próprio autor


```

-----
T E S T S
-----

Running br.com.alfa_tcc.test.CalculadoraTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.2 sec

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

```

Figura 12: Execução dos testes unitários

Fonte: Próprio autor

```

[calcweb-ec] $ /var/jenkins_home/tools/hudson.plugins.sonar.SonarRunnerInstallation/Scanner/bin/sonar-scanner -e -Dsonar.host.url=http://tcc-alfa:9000/
***** -Dsonar.language=java "-Dsonar.projectName=Calculadora Web" -Dsonar.projectVersion=1.0 -Dsonar.sourceEncoding=UTF-8 -Dsonar.projectKey=calcweb-1.0 -
Dsonar.java.binaries=/var/jenkins_home/workspace/calcweb-ec/target/classes -Dsonar.sources=src -Dsonar.projectBaseDir=/var/jenkins_home/workspace/calcweb-ec
INFO: Option -e/--errors is no longer supported and will be ignored
INFO: Scanner configuration file: /var/jenkins_home/tools/hudson.plugins.sonar.SonarRunnerInstallation/Scanner/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: SonarQube Scanner 3.2.0.1227
INFO: Java 10.0.2 Oracle Corporation (64-bit)
INFO: Linux 4.15.0-22-generic amd64
INFO: User cache: /root/.sonar/cache
INFO: SonarQube server 7.1.0
INFO: Default locale: "en", source code encoding: "UTF-8"
INFO: Publish mode
INFO: Load global settings
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.google.protobuf.UnsafeUtil (file:/root/.sonar/cache/3e26d9ec2d452f29e27358939e592296/sonar-scanner-engine-shaded-
7.1-all.jar) to field java.nio.Buffer.address
WARNING: Please consider reporting this to the maintainers of com.google.protobuf.UnsafeUtil
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
INFO: Load global settings (done) | time=87ms
INFO: Server id: AWV3MIEhpNYfjPxQKfYB
INFO: User cache: /root/.sonar/cache
INFO: Load plugins index
INFO: Load plugins index (done) | time=71ms
INFO: Load/download plugins

```

Figura 13: Execução da inspeção de código pelo SonarQube

Fonte: Próprio autor

```

Deploying /var/jenkins_home/workspace/calcweb-ec/target/calcweb-1.0.war to container Tomcat 8.x Remote with context calcweb-1.0
  Redeploying [/var/jenkins_home/workspace/calcweb-ec/target/calcweb-1.0.war]
  Undeploying [/var/jenkins_home/workspace/calcweb-ec/target/calcweb-1.0.war]
  Deploying [/var/jenkins_home/workspace/calcweb-ec/target/calcweb-1.0.war]
Finished: SUCCESS

```

Figura 14: Implantação automática da mudança

Fonte: Próprio autor

Ao final do processo, o Jenkins (vide Figura 15) exibe informações referentes aos projetos, tais como: situação, último sucesso, última falha e última duração. A situação é demonstrada utilizando dois ícones, uma esfera e uma imagem que faz analogia com a situação do tempo. A esfera, pode ser exibida na cor azul (ou verde, se utilizado o *plugin Green Balls*) em caso de sucesso, ou vermelha em caso de falha. Já no caso da imagem do tempo, vai depender dos resultados das últimas compilações, e pode oscilar desde um ícone com um sol radiante até uma nuvem carregada com raios.

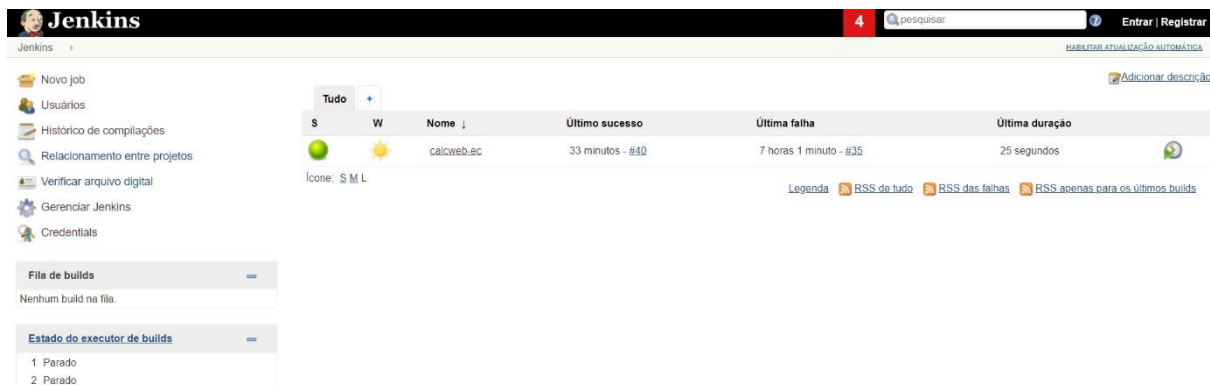


Figura 15: Dashboard dos projetos no Jenkins

Fonte: Próprio autor

Neste caso, como todas as etapas foram executadas com sucesso, basta atualizar a página do projeto CalcWeb, que será exibida a nova operação de potenciação (vide Figura 16), pronta para ser utilizada.

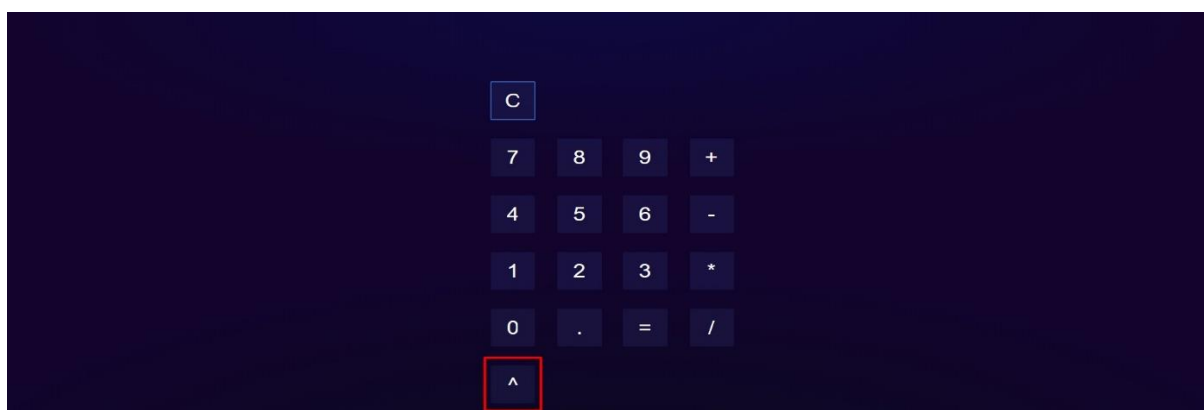


Figura 16: Tela principal do CalcWeb contendo as quatro operações básicas acrescida da potenciação

Fonte: Próprio autor

3.5 Material complementar

Visando facilitar a compreensão do cenário, foi produzido um vídeo contendo cada etapa descrita anteriormente. O material complementar está disponível no YouTube, por meio do seguinte *link*: <https://www.youtube.com/watch?v=m3kFQjoA-iQ&feature=youtu.be>.

CONCLUSÃO

Conforme demonstrado, a entrega contínua faz uso de um conjunto de conceitos, técnicas e ferramentas, formando um ciclo que contempla desde os registros das mudanças, desenvolvimento, controle de versões, integração contínua (processos automatizados: execução dos testes unitários, análise de código, compilação e construção), até a implantação das mudanças no ambiente de produção.

Por estar presente durante todo o ciclo de desenvolvimento, a utilização da entrega contínua é fundamental, pois traz uma série de benefícios, dentre os quais, destacam-se: colaboração entre os times de desenvolvimento e operações, redução da possibilidade da ocorrência de erros provenientes de falha humana e agilidade na geração das versões.

É importante ressaltar, que apesar das vantagens supracitadas, há algumas dificuldades que podem ser interpretadas, inicialmente, como desvantagens. Dentre os principais aspectos, destacam-se: custo de formação e manutenção de uma equipe especializada na área, curva de aprendizado e resistência dos demais membros da equipe. No entanto, são características e comportamentos inerentes a qualquer processo de mudança e que podem ser mitigados mediante a um gerenciamento adequado.

Portanto, pode-se concluir, embasado pela pesquisa teórica e demonstração prática, que é perfeitamente viável criar um *pipeline* de entrega contínua utilizando exclusivamente *software* livre. Diante disso, constatou-se que o resultado esperado foi alcançado, pois foi comprovado que é possível disponibilizar versões com mais qualidade, e com custos, tempo e riscos consideravelmente menores.

REFERÊNCIAS

ANICHE, Mauricio. *Test-Driven Development: Teste e Design no Mundo Real*. São Paulo: Casa do Código, 2012.

AQUILES, Alexandre; FERREIRA, Rodrigo. *Controlando versões com Git e GitHub*. São Paulo: Casa do Código, 2014.

GITHUB. *Mastering Issues*. 2014. Disponível em: <https://guides.github.com/features/issues/>. Acesso em: 02 jun. 2018.

HUMBLE, Jez; FARLEY, David. *Entrega contínua: como entregar software de forma rápida e confiável*. Porto Alegre: Bookman Editora Ltda., 2014.

HÜTTERMANN, Michael. *DevOps for developers*. New York: Apress, 2012.

JENKINS. *Jenkins User Documentation Home*. s.a.. Disponível em: <https://jenkins.io/doc/>. Acesso em: 07 abr. 2018.

JUNIT. *About*. s.a.. Disponível em: <https://junit.org/junit5/>. Acesso em: 07 abr. 2018.

MAVEN. *Apache Maven Project*. s.a.. Disponível em: <https://maven.apache.org/what-is-maven.html>. Acesso em: 07 abr. 2018.

MEDEIROS, Higor. *Gerência de Configuração e Mudanças*. 2014. Disponível em: <https://www.devmedia.com.br/gerencia-de-configuracao-e-mudancas/31327>. Acesso em: 26 ago. 2018.

MOLINARI, Leonardo. *Gerência de configuração: técnicas e práticas no desenvolvimento do software*. Florianópolis: Visual Books, 2007.

PRESSMAN, Roger S. Gestão de configuração de software. *In.*: PRESSMAN, Roger S. *Engenharia de Software: uma abordagem profissional*. 7. ed. Porto Alegre: AMGH Editora Ltda., 2011.

SAMPAIO, Cleuton. *Qualidade de software na prática: como reduzir o custo de manutenção de software com a análise de código*. Rio de Janeiro: Editora Ciência Moderna Ltda., 2014.

SATO, Danilo. *DevOps na prática: entrega de software confiável e automatizada*. São Paulo: Casa do Código, 2014.

SILVERMAN, Richard E. *Git: Guia prático*. São Paulo: Novatec, 2013.

SONARQUBE. *About SonarQube*. s.a.. Disponível em: <https://www.sonarqube.org/about/>. Acesso em: 11 ago. 2018.

CAMARGO, Thiago Oliveira. *Gerenciando a qualidade do código fonte com o SonarQube*. 2015. Disponível em: <https://www.devmedia.com.br/gerenciando-a-qualidade-do-codigo-fonte-com-o-sonarqube/32494>. Acesso em: 11 ago. 2018.

WAZLAWICK, Raul Sidnei. Gerenciamento de Configuração e Mudança. *In.*: WAZLAWICK, Raul Sidnei. *Engenharia de Software - Conceitos e Práticas*. Rio de Janeiro: Elsevier, 2013.