DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Automatização do Processo de Entrega de Softwares

Relatório submetido à Universidade Federal de Santa Catarina
como requisito para a aprovação da disciplina:
DAS 5511: Projeto de Fim de Curso

Vito Archangelo Giordani

Florianópolis, Julho de 2019

Automatização do Processo de Entrega de Softwares

Vito Archangelo Giordani

Esta monografia foi julgada no contexto da disciplina

DAS 5511: Projeto de Fim de Curso

e aprovada na sua forma final pelo

Curso de Engenharia de Controle e Automação

Prof. Carlos Barros Montez

Banca Examinadora:

Alex Sandro da Silva Pereira
Orientador na Empresa

Prof. Carlos Barros Montez
Orientador no Curso

Prof. Hector Bessa Silveira
Responsável pela disciplina

Prof. Tiago Semprebom, Avaliador

Maria Alice dos Santos Duz, Debatedora

Davinder Singh Chandhok, Debatedor

Agradecimentos

Agradeço infinitamente ao meu pai Luiz, minha mãe Teresina e meu irmão Emanuel por todo o apoio durante o curso, e também durante a vida.

Aos meus amigos, agradeço pela companhia, incentivo e suporte.

Por fim, agradeço à Universidade Federal de Santa Catarina e seu corpo docente por possibilitar a minha formação.

Resumo

As necessidades do mercado de software mudam rapidamente, visto que novas tecnologias são lançadas frequentemente e criam uma demanda de entrega rápida de novos sistemas que se adéquem a essas necessidades. Tradicionalmente, a elaboração de um software tem basicamente o seguinte fluxo: documentação dos requisitos, projeto do sistema, codificação, implementação e manutenção. A partir da adoção de métodos ágeis, que organizam o desenvolvimento de um software em pequenos ciclos, caso uma das etapas não consiga suprir a demanda, todo o fluxo de trabalho é afetado. Somando-se a isso, o processo de implementação, no qual o sistema é entregue para o cliente final, é executado tradicionalmente de forma manual, sendo, muitas vezes, um gargalo no fluxo de desenvolvimento. Portanto, este trabalho busca automatizar essa etapa, visando a redução do tempo necessário entre o fim de um ciclo de desenvolvimento e início de outro, possibilitando, então, entregas mais frequentes e respostas mais rápidas às necessidades do mercado. Os resultados obtidos mostram que a automação da implantação de softwares permite encurtar significativamente o tempo necessário entre o início do desenvolvimento de uma funcionalidade e sua entrega efetiva para o cliente.

Palavras-chave: Automatização. Implementação de software.

Abstract

The need of the software market changes quickly as new technologies are often introduced and create a demand for fast delivery of new systems that fit those needs. Traditionally, software development has the following flow: requirements documentation, system design, coding, implementation and maintenance. From the adoption of agile methods, which organize the development of software in short cycles, if one of the steps fails to meet the demand, the whole workflow is affected. In addition, the implementation process, in which the system is delivered to the end customer, is traditionally executed manually, often being a bottleneck in the development flow. Therefore, this work seeks to automate this step, aiming to reduce the time required between the end of one development cycle and the beginning of another, thus enabling more frequent deliveries and faster responses to market needs. The results show that the automation of the software implementation allows a significant shortening of the time needed between the beginning of the development of a functionality and its effective delivery to the client.

Keywords: Automation. Software deployment.

Lista de ilustrações

Lista de Códigos

Lista de tabelas

Lista de abreviaturas e siglas

API - *Application Programming Interface*

APT - *Advanced Packaging Tool*

CD - *Continuous Delivery*

CI - *Continuous Integration*

CPU - *Central Processing Unit*

DNS - *Domain Name System*

GB - *Gigabyte*

HTTP - *Hyper Text Transfer Protocol*

HTTPS - *Hyper Text Transfer Protocol Secure*

IP - *Internet Protocol*

RAM - *Random Access Memory*

SQL - *Structured Query Language*

SSH - *Secure Shell*

UFW - *Uncomplicated Firewal*

URL - *Uniform Resource Locator*

Sumário

1 Introdução

A cada dia que passa, novas empresas entram no mercado e competem por espaço e clientes. Um fator importante que define o sucesso ou fracasso dessas empresas é a capacidade e velocidade de se adaptar de acordo com as necessidades, mudanças, situações e circunstâncias. Para tanto, é essencial a habilidade de entregar rapidamente produtos que atendam às exigências do mercado. Porém, não basta agilidade, é preciso agregar valor ao produto por meio da sua qualidade.

Na indústria de software, muitos processos estão envolvidos desde o início da elaboração dos requisitos até a entrega final para o cliente e posterior manutenção. Uma etapa necessária, após finalizar o desenvolvimento do sistema, é a implementação dele, seja por meio da instalação em um servidor, envio para o cliente ou hospedagem em um site para ser acessado pela Internet. Como é tradicionalmente manual, esse processo é suscetível à automação, eliminando custos e reduzindo o tempo entre as entregas.

Além disso, com a ampla adesão de métodos ágeis para gerenciar projetos, é necessário diminuir o tempo necessário para entregar um sistema desenvolvido, visto que essa é uma etapa necessária para o fim do ciclo de desenvolvimento e inicio de outro. Ou seja, caso o desenvolvimento transcorra dentro do cronograma, porém a entrega apresente atrasos, todo o processo será afetado, pois o cliente só avalia o produto quando ele é entregue.

1.1 Justificativa

Atualmente, a disponibilização de um serviço é dependente da equipe de operações, a qual tem acesso aos servidores da empresa. Após o término da codificação de um software, o desenvolvedor envia o código para a equipe de operações e ela, por sua vez, gera o produto da compilação do código, que são os arquivos finais do sistema. Posteriormente, esses arquivos são enviados para um servidor previamente configurado e, então, a aplicação fica disponível para o acesso do cliente por meio da Internet.

Entretanto, são inúmeros os problemas decorrentes dessa abordagem. O primeiro surge quando o as configurações do servidor são diferentes das do ambiente de desenvolvimento. Isso gera a necessidade de retrabalho no sistema, a fim de adaptá-lo para rodar no ambiente de produção (servidor). Além disso, a equipe de desenvolvimento é quem tem um profundo conhecimento do sistema. Portanto, caso algum erro ocorra, é necessário alocar duas equipes para corrigir o sistema, a de desenvolvimento para recodificar e a de operações para disponibilizar o sistema corrigido.

Este trabalho, além de automatizar e reduzir o tempo de entrega de um software, funcionalidade ou correção, diminuirá a necessidade de alocar a equipe de operações para a etapa de disponibilização do sistema. Consequentemente, a equipe poderá focar no aprimoramento do ambiente dos servidores e na infraestrutura, por exemplo, e não somente atendendo às demandas de disponibilização de sistemas.

1.2 Objetivos

Este projeto tem como objetivo principal automatizar o processo de entrega de softwares por meio de ferramentas que executem o código fonte de aplicações, incluindo suas dependências, gerem um artefato executável o encaminhem para o destino final.

1.2.1 Objetivo Geral

Para atingir o objetivo de automatizar totalmente a entrega de sistemas, a partir do seu código fonte, serão necessários três ambientes: um para armazenar o código fonte das aplicações, outro para compilar esse código e transformá-lo em um executável e um terceiro para executar esse produto. Além disso, deverão ser analisadas as ferramentas disponíveis e escolhidas as que mais correspondam às necessidades do projeto.

1.2.2 Objetivos Específicos

A seguir serão elencados os objetivos específicos do projeto, que ilustram os passos necessários para atingir o objetivo geral.

- Configurar ferramentas para os seguintes propósitos:
  - Armazenamento do código;
  - Automatizar a compilação do software;
  - Armazenar o objeto resultante da compilação do software;
  - Automatizar a execução o objeto citado no item anterior;
  - Integrar todas as ferramentas configuradas no projeto.
- Atestar o funcionamento das ferramentas configuradas através de um estudo de caso.

1.3 Organização do Documento

Este documento está dividido em seis principais capítulos. A empresa na qual foi realizado o projeto é apresentada no Capítulo 2. O Capítulo 3 apresenta os conceitos essenciais para a compreensão do projeto.

Já a metodologia utilizada para possibilitar a automatização da entrega de um sistema é discutida no Capítulo 4. Primeiramente são avaliadas as etapas necessárias para desenvolver um sistema, desde a demanda do cliente até a entrega e manutenção. Em seguida, são apresentadas as ferramentas que serão configuradas em cada uma das etapas do projeto: armazenamento de código, automação de compilação, armazenamento de executáveis e implantação do sistema.

No Capítulo 5 é mostrado efetivamente como as ferramentas definidas no Capítulo 4 foram implementadas. Através de códigos com suas respectivas explicações e capturas de telas, são descritos todos os passos realizados no projeto.

Visando atestar o funcionamento dos ambientes configurados no Capítulo 5, no Capítulo 6 será mostrado um estudo de caso que aplica todas os passos para automatizar a entrega de um software. Para tanto, será mostrada uma aplicação que terá seu código fonte armazenado em um repositório, sendo posteriormente compilado e disponibilizado para o cliente final. Além disso, serão exibidos e explicados todos os arquivos necessários para possibilitar que o código fonte de qualquer sistema seja manipulado pelas ferramentas a fim de automatizar sua entrega.

Por fim, no Capítulo 7 será feito um apanhado geral do projeto, discutido os resultados globais e indicado se os objetivos foram atingidos, juntamente com uma avaliação de impactos causados pelo projeto na empresa.

2 A Empresa

A BRy Tecnologia, fundada em 2001 e localizada em Florianópolis - Santa Catarina, oferece soluções de segurança para documentos eletrônicos. Hoje, é a única empresa 100% brasileira a atuar na pesquisa, desenvolvimento e comercialização de soluções em certificação digital. Os produtos e serviços desenvolvidos estão presentes em autoridades certificadoras, empresas de software e empresas privadas, além de órgãos públicos estaduais e federais. Conta com mais de 15 mil clientes atendidos e mais de 1 bilhão de assinaturas digitais realizadas, fornecendo soluções na maioria das vezes em formato de produtos.

3 Fundamentação Teórica

Nesta seção serão explicados todos os conceitos necessários para a compreensão dos capítulos seguintes.

3.1 Métodos Ágeis

Basicamente, os métodos ágeis são um conjunto de práticas que buscam acelerar a entrega de um sistema. Além disso, eles promovem um gerenciamento de processos que possibilita a verificação dos resultados e adequações constantes.

Os métodos ágeis são cíclicos e incrementais, ou seja, ao final de cada ciclo são adicionadas novas funcionalidades ou retrabalhadas as que não apresentaram os resultados desejados. Ademais, o processo é dividido em pequenas funcionalidades, possibilitando a entrega delas ao final de cada ciclo, sendo que o cliente recebe o produto em etapas, com novas funcionalidades sendo adicionadas constantemente ao final de cada ciclo. Também é possível garantir um maior controle sobre os erros, visto que eles podem ser identificados consertados durante o desenvolvimento, minimizando o tempo necessário ao final do projeto para testes e correções de comportamentos inesperados.

Outro fator importante é a constante avaliação pelo cliente. Visto que a cada final de ciclo deve ser entregue um produto, caso seja necessário, o cliente tem a oportunidade de tomar decisões e alterar o projeto. Isso permite que o projeto seja adaptável, possibilitando a entrega de um produto mais fiel às necessidades do cliente.

Em acordo com a metodologia ágil, dezessete profissionais lançaram, em 2001, o manifesto ágil, que é uma declaração dos princípios que fundamentam o desenvolvimento ágil de softwares. Buscando melhores maneiras de organizar o desenvolvimento de softwares, foram definidos quatro valores essenciais:

- Os indivíduos e interações acima de processos e ferramentas;
- Software funcional acima de documentação completa;
- Colaboração com o cliente acima de negociação de contrato;
- Responder a mudanças acima de seguir um plano.

3.1.1 Scrum

Dentre os principais e mais utilizados métodos ágeis está o Scrum. Segundo Pries [1], durante o desenvolvimento de um produto, tanto o cliente quanto os desenvolvedores tendem

a adquirir novos conhecimentos sobre o produto no momento que as suas especificações começam a ser implementadas. Portanto, ambas as partes podem utilizar o Scrum para reduzir os atrasos e riscos causados por mudanças intermediárias nas especificações de um produto.

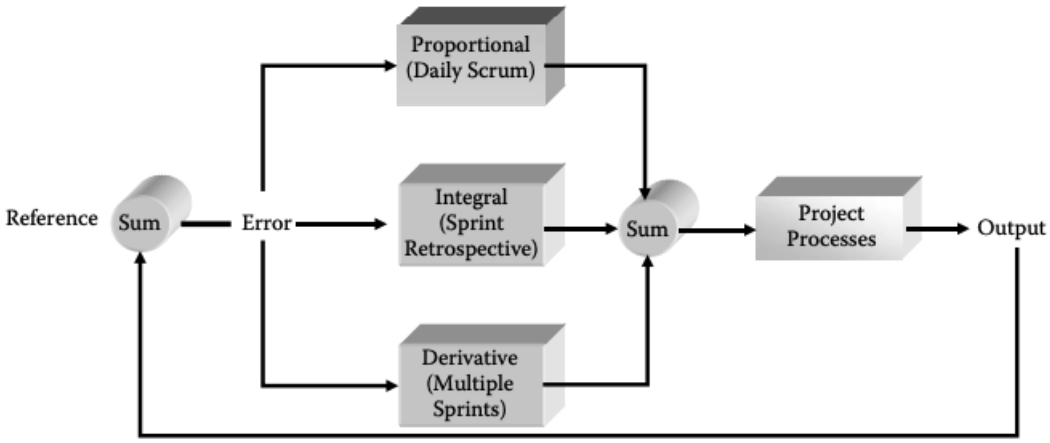
Seguindo os conceitos apresentados na seção 3.1, o equivalente aos ciclos dos métodos ágeis no Scrum são as *sprints*, que são intervalos nos quais um conjunto de atividades deve ser executado. Por sua vez, essas atividades são conhecidas como *product backlog*, que compreende uma lista de todas as funcionalidades que deverão ser adicionadas ao produto. Além disso, também são listados requisitos, comportamentos e alterações que deverão ser feitas no produto final.

De acordo com Schwaber e Sutherland [2], a *sprint backlog* é o apanhado de itens do *product backlog* selecionados para serem trabalhodos durante a *sprint*, além de um plano de entrega do produto incrementado ao final da *sprint*.

Ao final da *sprint* deverão ser apresentadas as funcionalidades implementadas, juntamente com a entrega do sistema incrementado para o cliente. Ele poderá dar seu parecer sobre a entrega e sugerir modificações que serão adicionadas ao *product backlog*.

O Scrum pode ser comparado a um sistema de controle em malha fechada, conforme mostra a Figura 1, no qual o sinal de saída, representado pelo produto entregue ao final de cada *sprint*, é comparado com uma referência, que é o produto final desejado. Essa comparação resulta em um erro que após passar pelos métodos teorizados pelo Scrum, é aplicado no projeto a fim de corrigi-lo.

Figura 1 – Scrum como Malha de Controle



Pries [1]

3.2 Versionamento de Código

De acordo com Somasundaram [3], um sistema de versionamento de código é uma aplicação capaz de gravar as mudanças em um ou mais arquivos durante um período de tempo, de modo que seja possível retornar a uma versão específica daquele arquivo em qualquer momento.

A grande vantagem possibilitada por esse tipo de sistema é a organização do projeto, visto que é possível manter um histórico do desenvolvimento, possibilitando desenvolver funcionalidades paralelamente a partir do mesmo código. Além disso, viabiliza a criação de uma nova versão do projeto sem alterar a versão principal.

Existem três tipos disponíveis de sistemas de controle de versão, ainda citando Somasundaram [3], classificados de acordo com seu modo de operação, sendo eles local, centralizado e distribuído. O local é capaz de manter um histórico dos arquivos armazenados no disco rígido do computador, somente, possibilitando alterar o conteúdo de um arquivo para qualquer estado rastreado. Porém, times de desenvolvimento não são capazes de trabalhar no mesmo projeto ao mesmo tempo, visto que os arquivos são armazenados em computadores individuais, sem acesso de terceiros.

Já no formato centralizado, os arquivos são armazenados em um servidor no qual todos os desenvolvedores têm acesso através dos seus computadores pessoais. Essa abordagem não somente possibilita acesso aos arquivos, como também oferece sobre o que está sendo alterado e quem efetuou isso. Além disso, como os arquivos são versionados em um único local, o servidor, qualquer mudança realizada nesses arquivos será automaticamente compartilhada com o restante do time também.

No formato distribuído, os desenvolvedores mantém todas as alterações nos seus computadores, sendo que essas máquinas - denominadas áreas de trabalho - podem se comunicar através de um servidor, que mantém outra cópia dos arquivos e seus respectivos históricos. Essa comunicação é feita por meio de dois comandos: o *pull*, que verifica todas as alterações do servidor e faz o download para a área de trabalho, e o *push*, que envia todas as alterações da área de trabalho para o servidor. Para tanto, é necessário efetuar um *commit* do conjunto de alterações nos arquivos, tornando-as permanentes e prontas para serem armazenadas no servidor através do *push*.

3.2.1 Git

Desenvolvido a partir de 2005 para apoiar a manutenção do *kernel* do Linux, o Git é um sistema distribuído de controle de versão de arquivos, que implementa todas as funcionalidades deste tipo de sistema, descritas na seção 3.2. Além disso, de acordo com Chacon [4], o Git possibilita a criação de *snapshots*, que são como fotografias do estado atual dos arquivos. Um *snapshot* é denominado *branch* e a partir dele é possível efetuar

alterações - e, inclusive, salvá-las no servidor - sem que o projeto principal seja modificado. Após finalizar as alterações é possível mesclá-las com o projeto original, originando uma nova versão do sistema.

3.3 Protocolo SSH

Sigla de *Secure Shell*, o SSH é um protocolo que fornece segurança para comunicação em redes. Segundo Barrett e Silverman [5], todos os dados transferidos entre computadores conectados por uma rede estão suscetíveis a manipulações por terceiros. Para evitar isso, o protocolo SSH automaticamente criptografa o dado na origem e descriptografa quando chega ao seu destino. Portanto, as duas partes participantes podem se comunicar sem o receio de que sua comunicação está sendo interceptada e sem se preocupar com os detalhes complexos da criptografia.

Possuindo uma fácil adesão, visto que é implementado em formato de um software, o protocolo SSH garante a autenticidade de informações repassadas, além do sigilo. Um uso comum desse protocolo é para acesso a máquinas remotas, visto que é possível implementá-lo através de uma senha cadastrada na máquina remota.

3.4 Contêineres

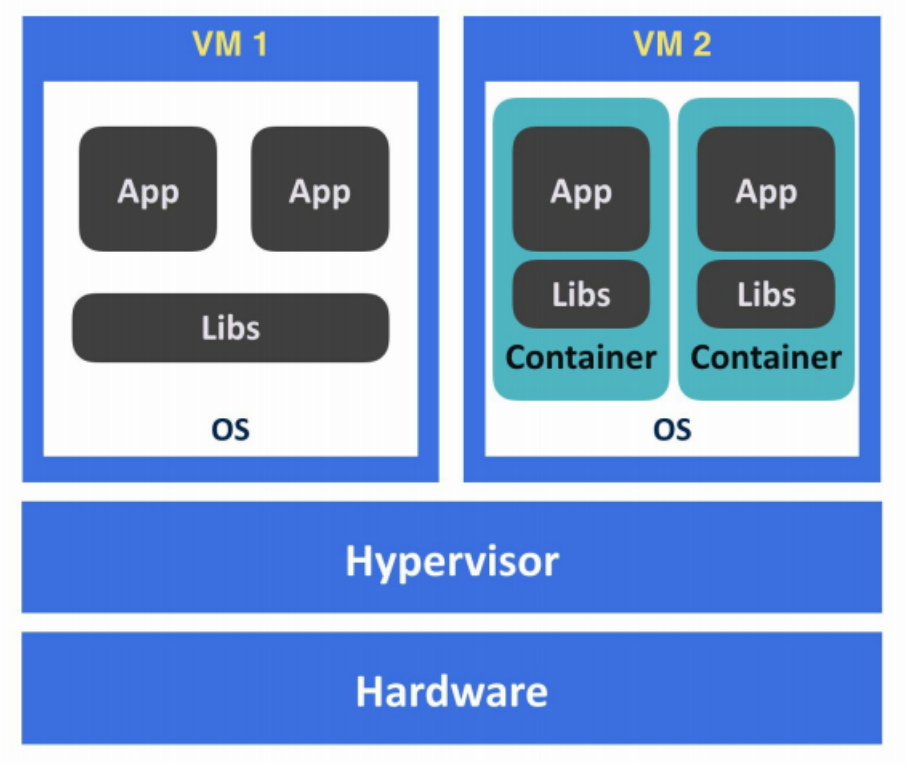
Quando uma aplicação é iniciada, ela consome CPU, memória RAM, espaço em disco e acesso à rede, basicamente. Caso várias aplicações coexistam na mesma máquina, elas consequentemente compartilharão os mesmos recursos, levando a uma situação de risco, visto que se uma aplicação causar um uso exagerado de recursos as outras serão afetadas. A tecnologia de contêineres vêm justamente para eliminar essa situação, proporcionando o gerenciamento de recursos através da execução de aplicações em ambientes isolados.

Os contêineres, no contexto de desenvolvimento de software, proporcionam o empacotamento de um aplicativo, juntamente com suas dependências, em uma imagem de contêiner. De acordo com a Microsoft [6], os aplicativos contidos em contêineres podem ser testados e implantados como uma instância de uma imagem de um contêiner em um sistema operacional, que por sua vez deverá suportar essa tecnologia.

Segundo Saito et al. [7], os contêineres possuem uma tecnologia similar com a das máquinas virtuais, visto que ambos fornecem uma camada de isolamento entre os aplicativos. Porém, diferentemente das máquinas virtuais, os contêineres compartilham o *kernel* do sistema operacional, necessitando de um espaço em disco significativamente menor. Em contrapartida, os contêineres englobam o aplicativo em conjunto com todas as suas dependências, diferentemente da abordagem da máquina virtual na qual as dependências são instaladas diretamente no sistema operacional. As diferenças entre máquinas virtuais

e contêineres podem ser melhor visualizadas na Figura 2. A máquina virtual indicada por "VM1"executa os aplicativos diretamente no seu sistema operacional, já a indicada por "VM2"faz uso da tecnologia de contêineres para criar um ambiente mais seguro e isolado.

Figura 2 – Arquitetura de Máquinas Virtuais e Contêineres



Fonte: Saito et al. [7]

Outra vantagem do uso de contêineres é a escalabilidade. Ou seja, é possível multiplicar o número de contêineres facilmente, visto que, a partir da imagem de um contêiner, é possível criar inúmeras instâncias da mesma aplicação rapidamente.

No escopo deste projeto, a maior vantagem fornecida é o isolamento de ambientes, já que uma aplicação, após ser contida em um contêiner, poderá ser instanciada em qualquer ambiente com suporte a contêineres a partir de sua imagem, evitando o recorrente problema do ambiente de implantação ser diferente do de desenvolvimento, gerando a necessidade de alocar equipes para adaptar a aplicação.

3.5 DevOps

Termo derivado da união das palavras "desenvolvimento"e "operações", representa uma prática que objetiva unificar o desenvolvimento de um software com a sua operação, etapa na qual o sistema é entregue para o cliente e recebe manutenções. De acordo com

Wahaballa et al. [8], práticas DevOps proporcionam uma redução do tempo do ciclo de produção de um software através da automação. É comum que o setor de desenvolvimento adote metodologias ágeis, conseguindo efetuar entregas frequentes e rápidas para atender as demandas recebidas, porém de nada adianta o desenvolvimento ser rápido se o setor operacional não é capaz de gerenciar todo o fluxo de entregas necessário.

Portanto, segundo Virmani [9], DevOps é um conjunto de princípios que guia o processo de elaboração de um software visando velocidade na entrega, testes contínuos e habilidade de reagir mais rapidamente às mudanças. Ou seja, DevOps estende os princípios das metodologias ágeis para a etapa de entrega do software.

Além disso, é importante citar dois conceitos fundamentais para a aplicação adequada das práticas de DevOps, sendo eles integração contínua e implantação contínua. A parte de integração refere-se a antecipar o máximo possível a etapa de compartilhar o código que está na máquina de um desenvolvedor com o resto do time de desenvolvimento. Posteriormente, esse código deverá ser validado através da sua adição ao código fonte da aplicação que está sendo desenvolvida, permitindo que o comportamento dessa nova funcionalidade, por exemplo, seja avaliado por todos os responsáveis pelo projeto, criando uma rotina na equipe de revisão das novas alterações.

O segundo conceito importante é relacionado à entrega do software, a etapa de implantação. Conforme Saito et al. [7], a implantação contínua é o ponto-chave do DevOps, visto que uma parte crítica no ciclo de desenvolvimento - não somente a codificação em si, mas todo o processo - do software. Como a entrega é tradicionalmente manual, os princípios de DevOps recomendam que ela seja automatizada através de ferramentas ou ambientas capazes de receber arquivos, contendo o código fonte de um sistema, por exemplo, e entregar aplicações em funcionamento.

4 Ferramentas

Antes de definir qual abordagem utilizar para atingir os objetivos do projeto, é preciso conhecer as etapas da entrega de um software e, então, verificar quais delas são suscetíveis à automação. O processo de desenvolvimento de software pode ser resumido na sequência das seguintes etapas:

1. Demanda do cliente;
2. Estudo de viabilidade;
3. Definição do cronograma;
4. Elaboração dos requisitos;
5. Codificação do projeto;
6. Testes do código;
7. Entrega do sistema para o cliente ou implementação do sistema na infraestrutura da empresa (*deploy*).

Os processos realizados nos itens 1 ao 5 são manuais e dificilmente podem ser substituídos por alguma automação, porém, os testes e o *deploy* são processos demorados, repetitivos e que requerem tempo. Portanto, o foco será a busca por ferramentas que possibilitem a automação das tarefas 6 e 7.

Uma possível abordagem para automatizar essas etapas é a codificação de um sistema que cumpra com todos os requisitos definidos. Esse sistema seria responsável por receber o código do desenvolvedor, junto com a indicação das bibliotecas e dependências utilizadas, e realizaria todas as tarefas necessárias para entregar o sistema em funcionamento, como testes e *deploy*. Porém, o processo de desenvolvimento de uma nova ferramenta é longo e complexo. Portanto, essa abordagem foi desconsiderada para a resolução do problema.

Levando isso em conta, foram buscadas outras formas de atingir os objetivos, evitando o uso de tempo e recursos desnecessários. Uma alternativa adequada foi utilizar ferramentas *open source* que permitissem a automatização da produção de softwares. Portanto, visando atingir as necessidades da empresa, foram utilizadas diversas ferramentas, onde cada ferramenta desempenha um papel único e específico na solução. Para tanto, os itens 6 e 7 devem ser divididos em subtarefas, a fim de facilitar a escolha das ferramentas mais adequadas. Essas subtarefas são mostradas na Figura 3 e exemplificadas a seguir.

1. No primeiro momento, após a equipe de desenvolvedores concluir a codificação de uma funcionalidade ou correção de um *bug* e finalizar os testes, o código da aplicação é enviado para um servidor. Esse servidor é responsável por armazenar arquivos e o histórico de suas alterações. Além disso, ele servirá como base para todo o processo de automatização, visto que qualquer alteração no código no servidor desencadeará as próximas etapas;

2. A automatização dos testes do código também é uma etapa necessária no projeto. Portanto, o segundo passo é testar a aplicação a partir de instruções escritas pelos desenvolvedores. O time de desenvolvedores receberá o *feedback* dessa etapa e a próxima etapa só será executada com o sucesso dos testes;

3. Posteriormente, a partir da última versão do código armazenado no servidor, será feita sua compilação. Nesse processo todas as dependências deverão ser embutidas no produto final, de modo que o resultado dessa etapa possa ser instanciado sem a necessidade de *downloads* de arquivos externos;

4. Visando eliminar a exigência de compilar o código toda vez que for criada uma nova instância da aplicação, foi identificada a necessidade de armazenar o produto da compilação, realizada na etapa 3. Esse armazenamento também deverá ser versionado e capaz de fornecer um histórico de versões.

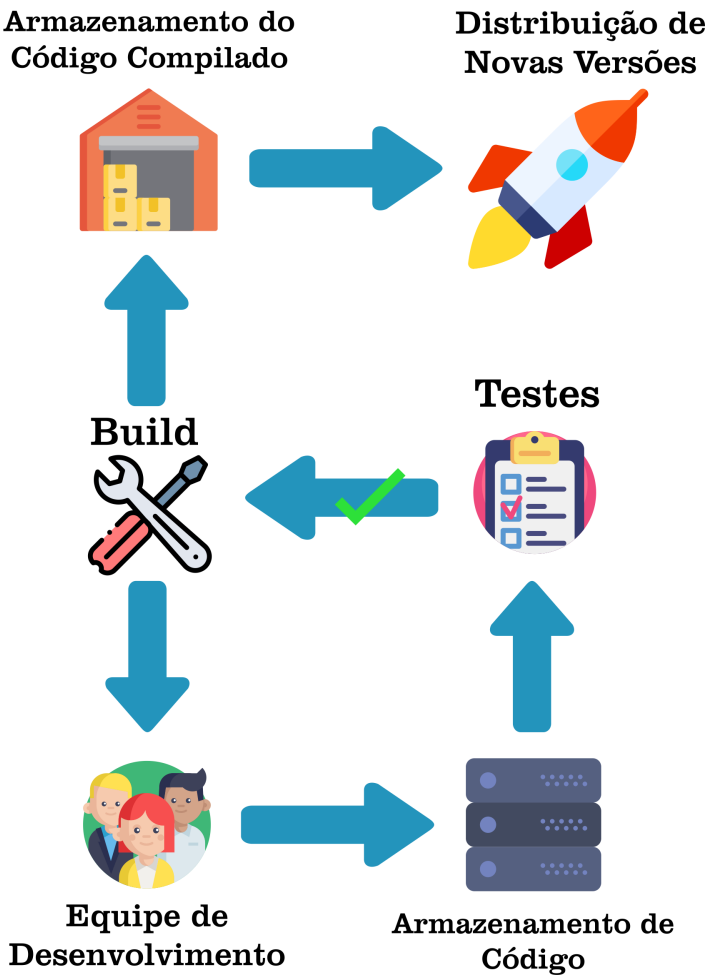
5. A última etapa do projeto é a distribuição do sistema para o cliente final. Ela será feita através da Internet, portanto as aplicações deverão conter configurações indicando como serão disponibilizadas.

Além disso, a equipe de desenvolvedores deve receber um *feedback* contínuo, não somente após a etapa de testes. Isso significa que se qualquer uma das etapas tiver um resultado indesejado, a equipe será notificada.

4.1 Contêineres Docker

Sempre que uma aplicação precisa transitar por vários ambientes - de uma máquina para outra, da máquina do desenvolvedor para uma máquina virtual que realiza os testes ou o *deploy* - como é o caso do sistema proposto neste projeto, há a dificuldade em configurar todos esses ambientes de forma idêntica para que a aplicação seja executada da mesma forma em todos eles. Portanto, a aplicação de contêineres no escopo desse projeto é essencial, visto que eles possibilitam uma abstração do sistema operacional, bibliotecas e dependências utilizadas. Isso é possibilitado por meio de uma tecnologia que cria um ambiente delimitado, no qual são inseridos os sistemas e todas as dependências que eles

Figura 3 – Fluxograma das Subtarefas



Fonte: Autor

necessitam para serem executados. Ou seja, um contêiner é um objeto que contém outros objetos.

Existem inúmeras soluções que permitem a utilização de contêineres. Porém, a solução mais popular é o Docker, que é uma ferramenta *open source* que possibilita a utilização de contêineres. Portanto, foi definido que os contêineres utilizados no projeto serão do Docker.

Os contêineres do Docker podem ser executados nativamente no Linux e no Windows. Ou seja, o desenvolvedor tem a liberdade para escolher qual sistema operacional será utilizado no seu ambiente de desenvolvimento, pois a partir do momento que a aplicação desenvolvida é empacotada num Docker, ela poderá rodar sem complicações em qualquer sistema operacional que suporte o Docker.

O empacotamento de uma aplicação em um contêiner do Docker gera uma imagem Docker. Logo, uma imagem é uma representação estática da aplicação com suas dependên-

cias e configurações, além do sistema operacional base. Quando surge a necessidade de rodar a aplicação contida em uma imagem Docker, seja para testes ou *deploy*, é instanciada um novo container com a imagem da aplicação.

No entanto, é necessário indicar como essas imagens deverão ser construídas. Isso é feito por meio de um arquivo denominado `Dockerfile`, que deverá ser versionado e armazenado junto com o código fonte da aplicação. Basicamente, é um arquivo de texto com instruções, comandos e passos que seriam executados manualmente para iniciar a aplicação, que será contida pelo contêiner originado desse `Dockerfile`. Ou seja, é um arquivo no qual é possível preparar o ambiente da aplicação a partir de um *script*. Através do comando `docker build` é possível automatizar a execução desses comandos e o produto final é uma imagem Docker pronta para instanciar a aplicação.

Por fim, as imagens do Docker devem ser armazenadas em um ambiente disponível para todos os desenvolvedores da empresa, denominado registro. A partir dessas imagens é possível gerar uma instância da aplicação em qualquer ambiente que suporte o Docker.

4.2 Armazenamento e Versionamento de Código

A etapa inicial da solução proposta é uma ferramenta de armazenamento e versionamento de código. Ela será a base para o funcionamento de todas as outras ferramentas, visto que não será armazenado somente os códigos das aplicações, mas também seus respectivos arquivos de configurações. Esses arquivos, por sua vez, serão utilizados em cada uma das ferramentas para definir como elas deverão se comportar e manipular o contêiner da aplicação. Por exemplo, as imagens de contêineres Docker são construídas a partir de instruções em um arquivo específico, o `Dockerfile`, que também deve ser versionado junto com o código fonte do sistema.

Como citado na seção 3.2, existem várias tecnologias para versionar código e dentro de cada uma delas há uma gama enorme de soluções. Levando isso em conta, foi definido que a tecnologia Git será utilizada, sendo necessário escolher uma solução que contemplasse todas as necessidades do projeto.

Um requisito essencial para essa ferramenta é a gratuidade e licença *open source*, a fim de possibilitar sua instalação na própria infraestrutura da empresa - visto que não é desejável nem adequado armazenar códigos de produtos da empresa em servidores de terceiros - além de não limitar o número de usuários. Além disso, a ferramenta deve ter disponível uma documentação abrangente e detalhada, com atualizações constantes. Outra característica, que talvez seja difícil de comparar entre as ferramentas disponíveis, é a atividade da comunidade em fóruns da ferramenta. Muitas vezes, ao trabalhar com alguma tecnologia, um erro ou situação bloqueante acontecem, e caso eles estejam documentados em algum fórum ou site de perguntas e respostas sua solução pode se tornar mais simples

e menos demorada.

Portanto, após analisar todas essas necessidades, a ferramenta definida para essa etapa foi o GitLab, que além de ser gratuita e *open source*, possui uma comunidade ativa, com atualizações constantes, extensa documentação e possibilita a instalação na infraestrutura da empresa.

Lançado em 2011, o GitLab possui funcionalidades de segurança, desempenho, gerenciamento e integração muito interessantes. No quesito segurança, um usuário administrador pode definir permissões específicas para cada usuário ou grupos de usuários. Por exemplo, a criação de projetos no repositório só será permitida aos líderes das equipes. Já os líderes têm a permissão de adicionar participantes aos projetos e definir suas respectivas permissões. Para exemplificar esse processo, pode ser criado um grupo de usuários para os líderes e outros grupos englobando os desenvolvedores de cada uma das equipes.

Além disso, ainda citando funcionalidades de segurança, a ferramenta possibilita acesso a diversos *logs* - forma como são chamados os registro de eventos - do sistema. É possível rastrear as informações de todas as requisições que chegam no servidor, como IP e cabeçalho da solicitação HTTP, além de *logs* das operações dentro do GitLab, como criação de usuários, projetos e operações de *pull* e *push*. Portanto, há uma grande possibilidade de auditar o sistema caso surja a necessidade. Outra função importante é a restrição do acesso ao servidor por faixa de IP. Portanto, caso o *login* e a senha de algum usuário sejam roubados, ainda é necessário realizar o acesso através da rede da empresa, criando uma barreira a mais de segurança.

O desempenho é uma questão importante na escolha de uma ferramenta, visto que deverá ser alocado um servidor que cumpra com os requisitos de recursos. O GitLab é executado com um uso de recursos consideravelmente baixo, levando em conta a complexidade e dimensão da ferramenta. Segundo a documentação oficial, os requisitos de *hardware* são os seguintes: CPU com `2 cores`, no mínimo, para até 500 usuários, e memória RAM com `8GB`, para até 100 usuários, quantidade consideravelmente superior ao número de funcionários da empresa. O espaço de armazenamento depende da quantidade de projetos e arquivos que serão inseridos no servidor, porém, é recomendável no mínimo `10GB` disponível em disco. Além disso, é possível monitorar quanto de cada um dos recursos citados está sendo efetivamente consumido.

4.3 Armazenamento de Imagens

Ficou definido na seção 4.1 que as aplicações deverão estar contidas em contêineres. Isso significa que a cada alteração do código da aplicação no GitLab deverá ser montada sua respectiva imagem Docker, a fim de possibilitar o teste e, posteriormente, o *deploy*. Ademais, armazenar todas as versões do sistema permite retornar o *deploy* a uma antiga

versão rapidamente caso alguma atualização, que não foi testada adequadamente, faça o sistema parar de funcionar. Essas imagens deverão ser armazenadas em um servidor, no qual roda uma aplicação, denominada registro, que tem por objetivo armazenar e gerenciar as imagens dos contêineres.

Do mesmo modo que nas outras etapas, para o armazenamento de imagens Docker há várias ferramentas disponíveis. A própria empresa desenvolvedora do Docker possui uma solução para este fim, o Docker Hub, que é um registro público de imagens. Porém, em algumas situações não é desejável tornar as imagens públicas, visto que elas usualmente contêm todo o código necessário para executar a aplicação, portanto o uso de um registro privado de imagens é essencial. Além disso, é indicado armazenar imagens base para utilizar nas imagens das aplicações. Por exemplo, ao invés de baixar as dependências da aplicação toda vez que sua imagem for montada, é possível criar uma imagem com essas dependências e utilizá-la como base no momento da criação da imagem da aplicação, economizando transferências de dados e, ainda, removendo a necessidade de criar uma conexão com o servidor que hospeda essas dependências, visto que, caso ele esteja indisponível, todo o processo de criar a imagem falha.

Dessa maneira, foi definido que essa etapa do projeto será desenvolvida com um registro *open source*, permitindo sua instalação e, consequentemente, o armazenamento das imagens na infraestrutura da empresa. Ademais, essa ferramenta ter suporte para qualquer tipo de tecnologia de contêiner - caso, futuramente, se deseje não utilizar mais o Docker para encapsular as aplicações.

Levando isso em conta, foi definido a ferramenta *Harbor Registry* como registro de imagens de contêineres, visto que ela atende a todas as necessidades citadas previamente. Uma característica muito interessante dessa ferramenta é que ela é disponibilizada em formato de imagens Docker. Ou seja, sua implementação é simplificada, visto que não é necessário configurar detalhadamente o servidor no qual ela será executada, bastando instalar o Docker, montar a imagem e definir as questões de acesso e armazenamento.

Outra característica importante é o registro de *logs* de absolutamente todas as operações realizadas no serviço. Não são armazenadas somente as operações na ferramenta em si, como a criação de usuários ou novos projetos, mas também as operações nas imagens, como o downloads ou uploads, permitindo o rastreamento de erros e falhas de segurança. Aliás, ainda falando sobre itens de segurança, é possível definir papéis e permissões para usuários ou grupos de usuários. Por exemplo, imagens só podem ser adicionadas por um grupo específico de usuários, enquanto outros só têm a permissão para fazer o download delas.

Por fim, a ferramenta disponibiliza uma API, acessada através de chamadas HTTP, que permite chamar funções da ferramenta sem usar sua interface gráfica. Isso possibilita implementar uma automação de processos, como por exemplo a criação de usuários. É

possível, através do uso dessa API, criar um usuário do *Harbor Registry* no mesmo instante em que for criado no GitLab, utilizando os mesmos dados e credencias de acesso.

4.4 Implementação de Softwares

Visando eliminar a dependência de configurações de sistema, na seção 4.1 foi definido que as aplicações serão executadas em contêineres Docker, portanto, deverão ser armazenadas no formado de imagens Docker. Isso restringe o ambiente de produção à execução apenas de contêineres Docker. Porém, essa restrição não afeta o funcionamento do ambiente, visto que, após ele ser configurado para executar esse tipo de contêiner, a partir de imagens armazenadas em um registro, qualquer aplicação que possua uma imagem Docker poderá ser executada nesse ambiente.

No entanto, o processo manual de *deploy* em contêineres é complexo, custoso e demorado. São necessários várias etapas entre o download da imagem, armazenada no registro, e a entrega da aplicação em funcionamento. Ainda, esse processo deve ser repetido para cada instância da aplicação, e em um ambiente de produção existirão vários contêineres espalhados em diversos *hosts*, visto que uma aplicação pode demandar múltiplos componentes, cada um com seu respectivo contêiner. Outra situação é a necessidade de garantir que os contêineres instanciados estejam sendo executados corretamente, por exemplo, se um contêiner cai, outro deve substituí-lo. Portanto, esta etapa do projeto é essencial para automatizar a entrega do software.

Existem inúmeras soluções para realizar o *deploy* de contêineres. Para o projeto foi definido que a ferramenta utilizada para automatizar essa etapa é o Kubernetes [10]. É uma ferramenta *open source*, que, além da automação do *deploy*, possibilita o dimensionamento e o gerenciamento de aplicações em contêineres. Dentre as inúmeras funcionalidades do Kubernetes, é possível destacar algumas delas que serão essenciais no desenvolvimento do projeto, explicadas a seguir.

Service Discovery: Caso as aplicações alvo do projeto necessitem ser entregues através de um servidor, é essencial que a ferramenta exponha o contêiner para a Internet, através de um endereço de IP ou DNS do domínio. Para tanto, o Kubernetes possui uma funcionalidade denominada *service discovery*, que permite externalizar o acesso aos contêineres.

Load Balancing: Como o número de acessos ao contêineres pode sofrer variações conforme a demanda da aplicação, é necessário que os *deploys* estejam preparados para tal situação. Para tal, o *load balancing* permite balancear a carga entre os contêineres, ou seja, caso o tráfego para uma determinada aplicação esteja muito elevado, é possível criar uma nova instância da aplicação e dividir o tráfego entre elas.

Rollouts and Rollbacks: Quando uma aplicação é atualizada, é necessário retirar de funcionamento os contêineres com a versão antiga que estão sendo executados. A fim de evitar a indisponibilidade da aplicação durante o período necessário para remover o contêiner atual e disponibilizar o contêiner com a nova versão, o Kubernetes fornece a ferramenta de *rollouts and rollbacks*, na qual contêineres com ambas as versões coexistem durante o processo de substituição. Portanto, os contêineres com as versões antigas só serão removidos a partir do momento que aqueles com a nova versão estejam disponíveis.

Self-healing: O Kubernetes possui um mecanismo de verificação da "saúde"dos contêineres, no qual, caso os contêineres não estejam no estado desejado eles são reiniciados. Para tanto, essa verificação é feita por meio de requisições definidas pelos desenvolvedores nos arquivos de configurações do Kubernetes.

Secret and Configuration Management: Essa funcionalidade é essencial para o projeto, visto que o Kubernetes precisará se autenticar nas outras ferramentas do projeto. Podem ser armazenadas informações sensíveis, como senhas e *tokens* de autenticação, em um ambiente seguro e controlado fora dos contêineres, evitando que eles tenham esses dados dentro deles. Somando-se a isso, é possível alterar esses *secrets* sem a necessidade de instanciar novamente os contêineres que os utilizam.

4.4.1 Componentes

A fim de possibilitar a compreensão dos tópicos abordados a seguir, é necessário conhecer os principais conceitos e componentes sobre a ferramenta Kuberntes, que serão explicados a seguir.

- ***Master***: Componente que executa os processos de controle do Kubernetes. Ele é a porta de entrada da ferramenta e é composto por diversas ferramentas que permitem definir o seu comportamento. Além disso, o *master* é responsável por realizar a comunicação e distribuir o trabalho entre os *nodes*.

- ***Nodes***: Um *node* pode representar uma máquina física, virtual, um montante de recursos alocados em algum provedor de computação em nuvem, etc. É um conceito abstrato que possibilita eliminar preocupações com as peculiaridades dos hardwares. Desse modo, um *node* pode ser visto como um conjunto de recursos - como CPU e RAM - disponíveis para uso.

- ***Cluster***: É um grupo de *nodes*. Adiciona uma camada de abstração maior, agrupando máquinas para representar somente os recursos delas.

- ***Pod***: É menor e mais simples objeto do Kubernetes. Ele fica dentro de um *node* e agrupa um ou mais contêineres em execução.

4.4.2 Arquitetura

Além das principais funcionalidades e dos componentes, é necessário conhecer a arquitetura da ferramenta a fim de compreender os passos para suas configurações e uso, descritos nos próximos capítulos.

Basicamente, há dois principais componentes no Kubernetes: o *master* e os *nodes*. O *master* é responsável pelas atividades gerenciais, como distribuição de *pods* entre os *nodes* disponíveis. Além disso, ele toma as decisões sobre o *cluster*, detectando e respondendo aos seus eventos, criando novas instâncias de pods quando alguns estão com desempenho insatisfatório, por exemplo.

Já os *nodes*, descritos na subseção 4.4.1, são incumbido de processar as requisições do *master*, ou seja, executar os *pods* contendo os contêineres com as aplicações. Ainda, dentro desses dois componentes principais existem outros diversos subcomponentes, que são mostrados na Figura 4 e explicados a seguir.

Dentre os componentes do *master* os principais são:

- **API *Server***: Porta de entrada do Kubernetes que provém comunicação para o meio externo e interno. É uma interface que dispõe de todos os recursos necessários para manipular os elementos do Kubernetes. Todas e quaisquer comunicações entre os componentes do *master* e os dos *nodes* são realizadas através do API *Server*. Visto que o *master* e os *nodes* não estão no mesmo ambiente, o API *Server* geralmente é configurado para escutar conexões criptografadas remotas no protocolo HTTPS.

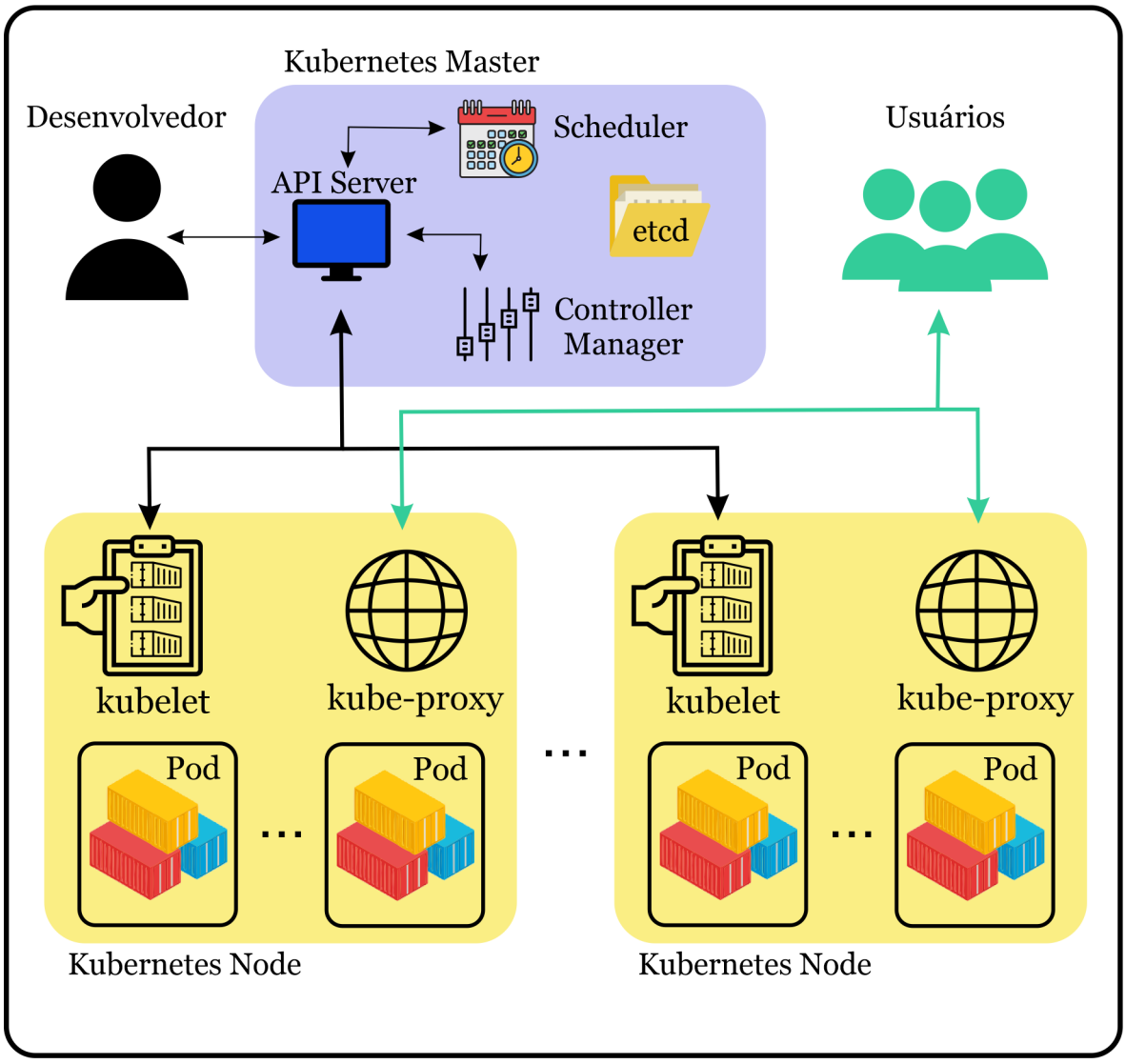
- ***Scheduler***: É o principal responsável por alocar os contêineres nos seus devidos lugares, selecionando em qual *node* será colocado um novo *pod*. Para tanto, ele mapeia os recursos distribuídos pelos *nodes* a fim de garantir que cada *node* receba somente a quantidade de trabalho que consegue suportar. Portanto, o *scheduler* deve conhecer os recursos disponíveis e os necessários para alocar cada *pod*.

- ***Controller Manager***: Esse componente fica constantemente identificando o estado dos *cluster*, através da API *Server*, e atua quando o estado identificado é diferente do desejado.

- **etcd**: É um serviço de armazenamento configurado especialmente para as especificidades do Kuberntes. Armazena dados no formato chave-valor, que podem ser configurações do *cluster*, representações do estado do *cluster*, como quais *nodes* estão em qual *cluster*, qual *pod* deveria estar sendo executado, qual contêiner está em execução, entre outros.

Já nos componentes dos *nodes*, além dos *pods* descritos na subseção 4.4.1, é possível citar:

Figura 4 – Arquitetura do Kubernetes



Fonte: Autor

- **kubelet**: Um agente que é executado em cada nó no *cluster*. Ele garante que os contêineres são executados nos *pods*. Além disso, garante que esses contêineres estão com o funcionamente adequado e desempenhando as funções para as quais foram criados.
- **kube-proxy**: Responsável por gerenciar o acesso aos *nodes* através da rede.

Porém, é necessário indicar como as aplicações deverão ser distribuídas através dessa ferramenta. Isso é permitido através de uma combinação de arquivos com instruções e comandos do `kubectl` - que nada mais é que a interface da API *Server* acessada através da linha de comando. Por exemplo, o comando `kubectl apply -f deployment.yaml` inicia a aplicação configurada no arquivo `deployment.yaml`, que indica como ela deverá

ser disponibilizada, qual imagem utilizar para instanciar os contêineres, entre outras informações.

4.5 Integração das Ferramentas

Nas seções anteriores foram definidas e explicadas todas as ferramentas necessárias para cada etapa da automação da entrega de um software. Porém, essas ferramentas isoladas umas das outras não têm muita utilidade, portanto é essencial integrá-las.

O GitLab, descrito na seção 4.2, possui uma ferramenta capaz de realizar essa tarefa, denominada GitLab CI/CD. A sigla CI vem de *Continuous Integration*, que se refere à integração do código desenvolvido por vários profissionais, realizando testes após a integração automática. Já a sigla CD, de *Continuous Delivery*, é referente ao processo de entrega (*deploy*) continuo, que automaticamente coloca em produção o código integrado na etapa do CI. Existem outras soluções que cumprem os requisitos de integração das ferramentas, porém como o GitLab já será utilizado é vantajoso adotar sua própria ferramenta de integração.

Para utilizar o GitLab CI/CD são necessários dois passos: adicionar um arquivo `.gitlab-ci.yml` no repositório do projeto, junto com o código fonte, e configurar um *runner* para executar as instruções desse arquivo. Um *runner* pode ser representado por uma máquina, virtual ou não, apesar de ser um elemento um pouco diferente que será explicado posteriormente. Então, cada novo *commit* desencadeará uma tarefa no *pipeline* da integração contínua.

O arquivo `.gitlab-ci.yml` define o que será executado, como será e o que ocorrerá caso situações específicas ocorram, como falha ou sucesso de um processo. Ele é dividido em etapas, e cada uma dessas etapas está associada às ferramentas descritas anteriormente. Se todas as etapas terminarem com sucesso, o *commit* é marcado com um ícone verde simbolizando que as alterações não causaram nenhum comportamento inesperado.

5 Configuração dos Ambientes

Nesta seção será mostrado como foram configurados os ambientes, as ferramentas e os passos necessários para deixar todas elas funcionais. Além disso, será mostrado como configurar a ferramenta depois de instalada, por exemplo: cadastro de novos desenvolvedores no GitLab, adição de uma nova máquina para execução dos contêineres no Kubernetes. Primeiramente será explicado o ambiente no qual foram instaladas as ferramentas, mostrando os passos necessários para deixá-los prontos para receber tráfego externo, para depois mostrar os passos necessários para configurá-las.

5.1 Máquinas Virtuais

Para executar as ferramentas descritas no Capítulo 4 é necessário um ambiente apropriado, com as configurações e recursos especificados. Uma forma de entregar os ambientes requisitados é por meio de máquinas virtuais. Elas são disponibilizadas por softwares que a partir de arquivos, denominados "imagem", conseguem fornecer um sistema operacional em funcionamento. De modo geral, é um computador dentro de um computador. A máquina virtual é isolada da máquina física que está a hospedando, impossibilitando que os softwares executados e arquivos armazenados dentro de uma máquina virtual entrem em contato com a máquina física ou com outras máquinas virtuais. Em relação aos recursos, cada máquina virtual conta com seus próprios hardwares, como memória, CPUs e discos rígidos. Portanto, os hardwares virtuais são mapeados para os hardwares físicos da máquina hospedeira, o que possibilita economia de recursos, reduzindo a necessidade de hardwares específicos.

Além disso, como as máquinas virtuais são armazenadas em arquivos, uma característica importante para o projeto é a possibilidade de criar uma cópia desses arquivos. Isso possibilita que, caso a máquina virtual sofra uma falha crítica, é possível retornar seu estado a um anterior à falha, evitando a perda de dados extremamente valiosos, como os códigos dos softwares desenvolvidos pela empresa. Outra possibilidade é alterar as características do sistema operacional hospedeiro, caso o uso das ferramentas cresça de modo que as configurações de hardware atuais não consigam lidar com o número de requisições.

Analisando o escopo do projeto, é possível configurar todas as ferramentas na mesma máquina virtual, porém, para isso seria necessário uma única máquina com configurações de hardware mais pesadas. Além disso, os hardwares de armazenamento, como o disco rígido, seriam compartilhados, e em caso de falha seria necessário retornar a um estado previamente salvo de todas as ferramentas. Em contrapartida, é possível configurar uma máquina

Tabela 1 – Recursos das Máquinas Virtuais

	Recursos		
Ferramenta	**RAM [GB]**	**CPU [Núcleos]**	**Espaço em Disco [GB]**
GitLab	8	2	40
Harbor Registry	4	2	40
Kubernetes Node	2	1	5
GitLab Runners	2	1	5

virtual específica para cada ferramenta, evitando as situação descritas anteriormente. Essa abordagem fornece mais segurança e maior grau de customização, visto que a máquina virtual será configurada especificamente para cada ferramenta, atendendo todas - e somente - suas necessidades. Somando-se a isso, a separação das ferramentas em diferentes máquinas virtuais garante que caso a máquina virtual que está executando uma ferramenta sofra uma falha fatal, somente uma parte do processo será afetada, não interrompendo totalmente fluxo do *pipeline*.

Portanto, para o projeto serão configuradas quatro máquinas virtuais, uma para cada ferramenta. A Tabela 1 detalha os recursos alocados para cada uma delas. O sistema operacional definido para todas elas é o Ubuntu Server 18.04, visto que é uma distribuição gratuita e configurado especificamente para atuar como servidor, sem interface gráfica e com baixo uso de recursos. Além disso, possui uma ampla biblioteca com vários softwares para download. As máquinas virtuais do "Kubernetes Nodes"e "GitLab Runnes"possuem requisitos de recursos mais modestos pois não será executado nenhum software com grande necessidade de recurso neles. Eles servirão para executar as aplicações e integrar as ferramentas, respectivamente.

5.1.1 *Servidor Web*

As máquinas virtuais atuarão como servidores que serão acessados através de uma conexão com a Internet. Para tanto, o sistema operacional deverá ser configurado a fim de permitir uma conexão segura e criptografada. Todas as máquinas virtuais serão acessadas remotamente através do protocolo SSH, descrito na seção 3.3, portanto será necessário configurá-lo. Esse tipo de acesso permite executar instruções dentro da máquina virtual através da linha de comando.

As configurações descritas nesta seção foram aplicadas em todas as máquinas virtuais utilizadas no projeto. Primeiramente o sistema operacional Ubunbtu Server 18.04 foi instalado. Posteriormente, foi necessário utilizar ferramentas para criar um servidor habilitado a receber tráfego pela Internet e configurar o acesso seguro a ele.

5.1.2 Protocolo SSH

A configuração da comunicação pelo protocolo SSH é simples, visto que existe uma ferramenta no repositório padrão do Linux que implementa essa configuração, chamada OpenSSH [11]. Após instalar essa ferramenta é necessário permitir o acesso ao servidor pela porta utilizada pelo protocolo SSH (22). Para tanto, foi utilizada a ferramenta `UFW` [12], que gerencia o *firewall* do servidor, sendo uma ferramenta nativa do servidor não é necessário sua instalação. Uma prática de segurança adotada foi a permissão de acesso às máquinas virtuais apenas por endereços de IP específicos.

5.1.3 Servidor

Para acesso ao servidor por parte do cliente foi configurada a ferramenta Nginx. Ela atua como uma porta de entrada, lidando com todas as requisições recebidas pelo servidor e redirecionando-as para as rotas definidas. Também possibilita instalar certificados SSL, que são empregados para fornecer uma comunicação criptografada entre o cliente e o servidor.

Portanto, foi necessário instalar a ferramenta no servidor, etapa realizada utilizando o `apt` do Ubuntu. Após instalar o Nginx é necessário permitir o acesso às portas dos protocolos HTTP (80) e HTTPS (443). A porta 80 só servirá para direcionar o tráfego recebido para a porta 443, que implementa protocolos de segurança e criptografia. A Figura 5 mostra o acesso ao servidor após sua configuração inicial.

É importante notar que a conexão é indicada como não segura. Isso se deve à ausência de um certificado SSL configurado para criptografar as conexões, levando à impossibilidade de acessar o servidor através do protocolo HTTPS. Portanto, o próximo passo é gerar um certificado SSL para criptografar as conexões. Além disso, o servidor somente pode ser acessado através do IP, visto que não há um domínio atribuído a ele em uma lista DNS.

5.1.3.1 Protocolo HTTPS

Para implementar uma conexão segura é necessário criptografar os dados transmitidos, processo que é possibilitado através do uso de certificados SSL. Para tanto, primeiramente foi requisitado um domínio em um provedor gratuito para fins de testes, que posteriormente será substituído pelo domínio da empresa.

A emissão do certificado SSL está atrelada ao domínio, ou seja, o certificado só encriptará requisições feitas para o domínio para o qual ele foi emitido. Portanto, foi emitido um certificado pela autoridade de certificação Let's Encrypt [13], que fornece certificados gratuitamente e possui uma ferramenta que possibilita automatizar a emissão desses certificados, porém, esse processo não foi aplicado no projeto. O certificado foi

Figura 5 – Configuração Incial do Servidor Nginx

Fonte: Autor

gerado utilizando a ferramenta `certbot` no Linux. Para tanto, foi necessário comprovar que o domínio indicado na requisição do certificado é controlado por quem o requisitou.

Após configurar o servidor Nginx com o certificado gerado é possível acessá-lo utilizando uma conexão segura através do protocolo HTTPS, como mostra a Figura 6. Posteriormente, quando as ferramentas forem instaladas em cada uma das máquinas virtuais, o tráfego recebido pelo Nginx será direcionado para as ferramentas, porém a camada de segurança será gerenciada pelo servidor Nginx.

Posteriormente, o prefixo `nginx` será substituído pelo nome da ferramenta sendo executada em cada máquina virtual.

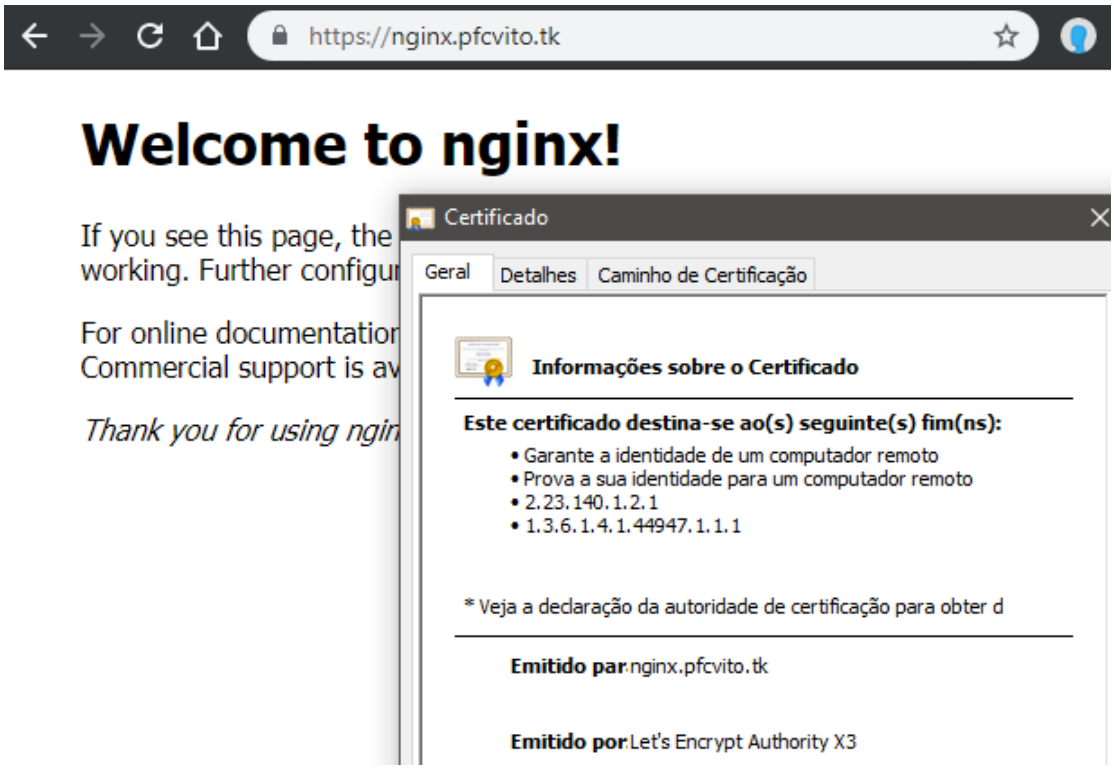
5.2 GitLab

Após configurar uma máquina virtual com os recursos necessários, conforme mostrado na Tabela 1, é necessário instalar a ferramenta no ambiente. Para tanto, o GitLab [14] disponibiliza vários métodos de instalação. Para o Ubuntu, é possível instalar em contêineres Docker ou diretamente no sistema operacional. Como a máquina virtual foi instanciada exclusivamente para hospedar o GitLab, a última opção foi a escolhida.

Primeiramente, já que GitLab disponibiliza arquivos binários da aplicação prontos para uso, foi necessário adicionar ao repositório de pacotes do Ubuntu, o `apt`, o repositório do GitLab. Isso possibilitará o download da ferramenta com apenas um comando.

A instalação da ferramenta é realizada no mesmo processo que o download, através da execução comando `apt install gitlab-ce` com privilégios de administrador. Após essa etapa foi preciso editar o arquivo de configurações do GitLab, mostrado no Código 1,

Figura 6 – Configuração HTTPS do Servidor Nginx



Fonte: Autor

e posterior reinicialização.

Código 1 – Configurações do GitLab

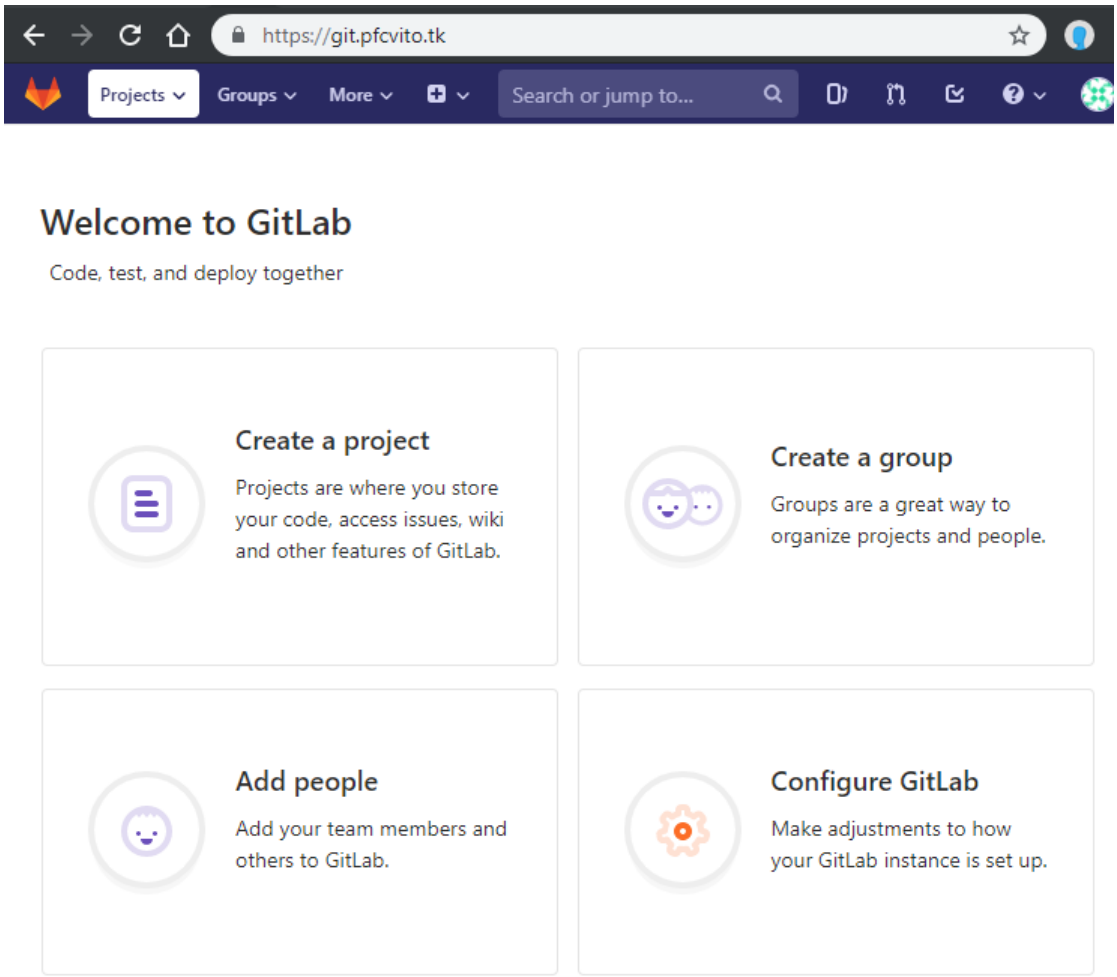
```
1 external_url 'https://git.pfcvito.tk'
2 nginx['redirect_http_to_https'] = true
3 nginx['ssl_certificate'] = "/etc/letsencrypt/live/pfcvito.tk/
    fullchain.pem"
4 nginx['ssl_certificate_key'] = "/etc/letsencrypt/live/pfcvito.tk
    /privkey.pem"
5 nginx['ssl_protocols'] = "TLSv1.1 TLSv1.2"
```

É possível visualizar no arquivo editado a indicação do domínio e do redirecionamento do tráfego da porta 80 (HTTP) para a porta 443 (HTTPS), a fim de evitar conexões inseguras. Além disso, os certificados gerados na subseção 5.1.3.1 são utilizados para criptografar a conexão.

Após configurar adequadamente a ferramenta, é possível acessá-la pelo link definido na `external_url`. No primeiro acesso é requisitada a definição de uma senha para o usuário administrador. A Figura 7 mostra a tela inicial, após o *login*. A partir desse momento a ferramenta está configurada e funcional, permitindo a criação de novos usuários

e projetos. Além disso, é possível desabilitar o cadastro de novos usuários, evitando assim que a ferramenta seja acessada por usuários não permitidos.

Figura 7 – Tela Inicial do GitLab



Fonte: Autor

5.3 Harbor Registry

Essa ferramenta será responsável por armazenar as imagens Docker de todas as aplicações que passem pelo fluxo das ferramentas configuradas. Assim como o GitLab, o Harbor Registry [15] é distribuído em arquivos binários compilados e prontos para o uso, bastando apenas configurar um arquivo que indica como ele deverá funcionar.

Após configurar a máquina virtual, com os requisitos descritos na Tabela 1, e emitir um certificado SSL para o domínio da ferramenta, processo descrito na seção 5.1, foi feito o download do pacote Harbor diretamente do seu repositório de código. Como o Harbor é executado em contêineres Docker, também foi necessário instalar o Docker na máquina virtual. Além disso, não é requerido instalar nenhum sistema de gerenciamento

de banco de dados, visto que o utilizado pela ferramenta - PostgreSQL - possui a imagem do seu contêiner atrelada às imagens do Harbor. Porém, devido à volatilidade intrínseca aos contêineres, é necessário indicar um diretório - no sistema operacional que hospeda o Docker - no qual serão armazenados arquivos de dados permanentes. Portanto, ao excluir ou reiniciar um contêiner, basta indicar esse diretório como um volume de dados, evitando a perda de dados.

Dentro do pacote de instalação do Harbor Registry, além das imagens Docker da ferramenta, está contido um arquivo `harbor.yml`, no qual são definidas todas as configurações. Esse arquivo é mostrado no Código 2, indicando a habilitação das portas 80 (HTTP) e 443 (HTTPS), além do caminho para o certificado SSL emitido. Também são definidas as senhas iniciais para acesso ao portal do Harbor e ao banco de dados, que posteriormente deverão ser alteradas. A *tag* `data_volume` indica o diretório da máquina virtual no qual será armazenado os dados da ferramenta. Já na *tag* `log` pode-se indicar o diretório da máquina virtual no qual serão armazenados os arquivos de *logs*, visto que se alguma falha ocorrer durante a execução do contêiner, todos os dados contidos nele não serão salvos.

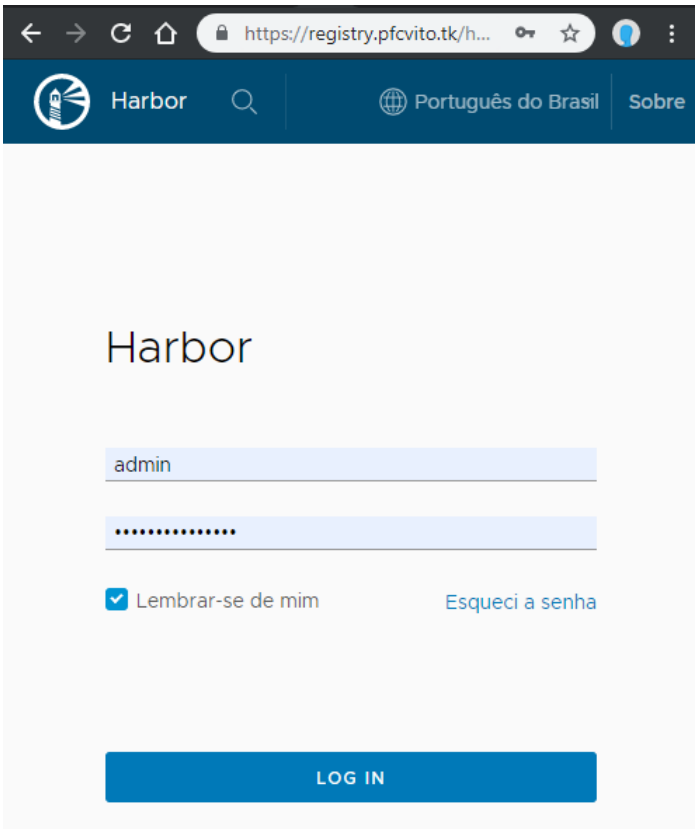
Código 2 – Configurações do Harbor Registry

```
1  hostname: registry.pfcvito.tk
2  http:
3    port: 80
4  https:
5     port: 443
6     certificate: /etc/letsencrypt/live/pfcvito.tk/fullchain.pem
7     private_key: /etc/letsencrypt/live/pfcvito.tk/privkey.pem
8  harbor_admin_password: Harbor12345
9  database:
10   password: root123
11 data_volume: /data
12 log:
13   level: info
14   location: /var/log/harbor
```

Após configurar a ferramenta e instanciar seus contêineres, é possível acessá-la pelo link definido na *tag* `hostname`, passo mostrado na Figura 8. Além disso, é necessário redefinir a senha de administrador indicada no arquivo `harbor.yml`.

Também é possível definir qual tipo de *log* se deseja obter. Existem cinco níveis de gravidade de *logs*, sendo que ao habilitar um tipo, os *logs* de gravidades subsequentes também serão habilitados. A seguir é mostrada a lista dos possíveis tipos de *logs*, listados crescentemente em ordem de gravidade. Para o projeto foi definido que serão mostrados os

Figura 8 – Tela Inicial do Harbor Registry



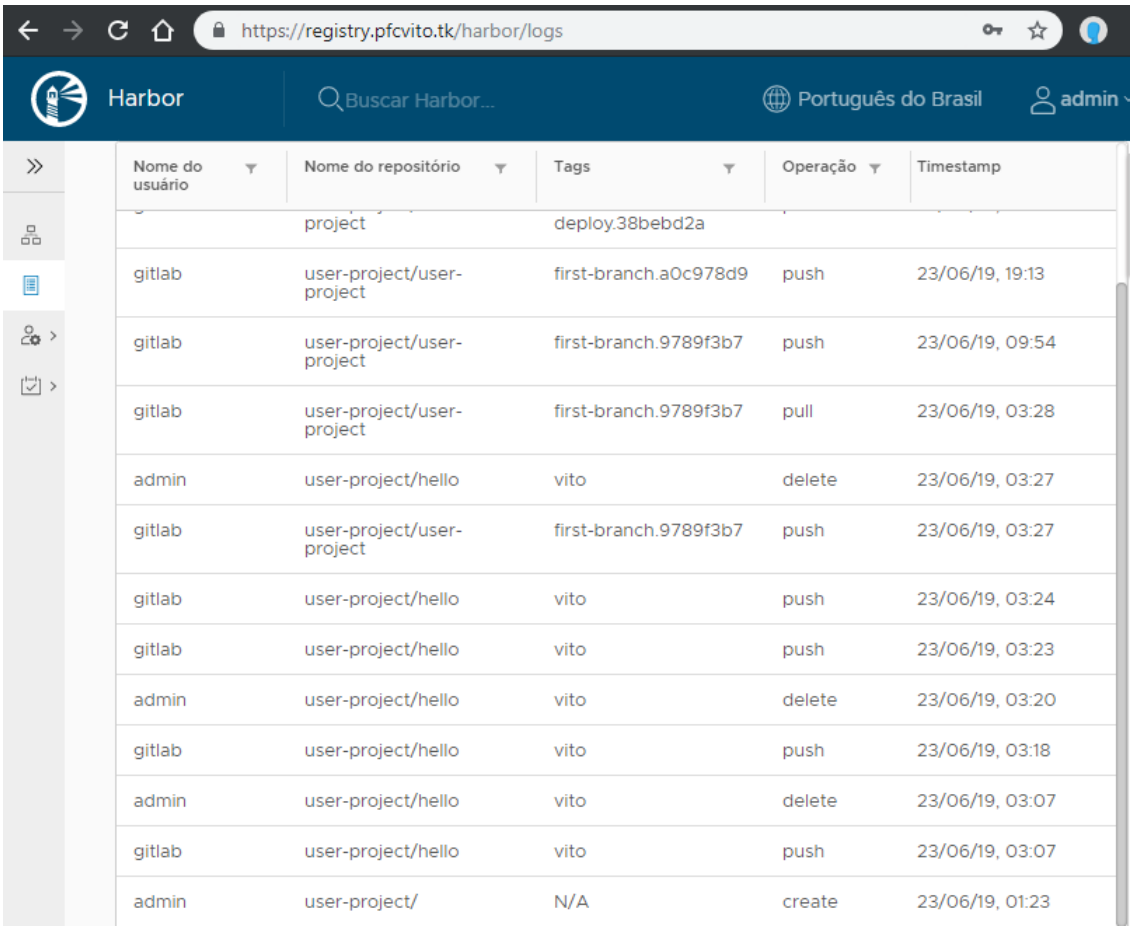
Fonte: Autor

logs classificados com gravidade "Info"acima. Isso possibilita rastrear todas as operações realizadas dentro do Harbor, como criação de usuários e *upload* e *download* de imagens. A Figura 9 mostra os logs obtidos durante os testes da ferramenta, como criação de projetos, usuários e operações com imagens.

1. *Debug*: Informações associadas ao *build* da ferramenta, geralmente utilizadas por desenvolvedores para depurar execuções do sistema;
2. *Info*: Informações de operações realizadas no sistema, como criação e *login* de usuários, ações no repositório de imagens;
3. *Warning*: Situações que causam pouco ou nenhum dano ao sistema;
4. *Error*: Erros na execução do fluxo normal do sistema, porém sem interrompimento no funcionamento;
5. *Fatal*: Erros que impedem o funcionamento normal do sistema.

Como administrador, é possível gerenciar projetos, usuários, repositórios e réplicas. Esta última é útil caso seja necessário portar as imagens do registro configurado para outro

Figura 9 – Registro de Logs do Harbor Registry



Fonte: Autor

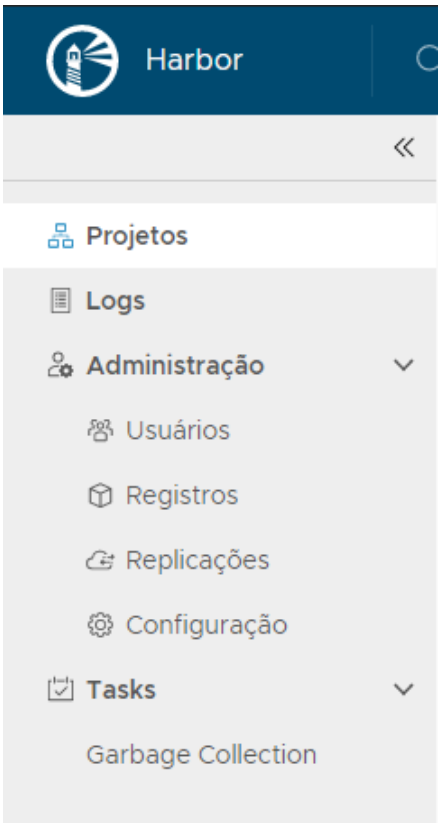
registro, como uma cópia de segurança. Além disso, assim como no GitLab, é permitido desabilitar o cadastro de novos usuários, atribuindo essa permissão somente aos usuários administradores da ferramenta. Essas opções são mostradas na Figura 10.

5.4 Kubernetes

O Kubernetes será responsável por gerenciar os contêineres das aplicações que passam pelo *pipeline* do sistema. Como descrito na seção 4.4, são necessários dois componentes para configurar o Kubernetes: o *master* e os *nodes*. O projeto utilizará somente um *node*, localizado em uma máquina virtual, descrita na Tabela 1.

Para simplificar a elaboração do projeto, foi utilizado um provedor privado de computação em nuvem para hospedar o *master*. Como os *nodes* podem ser quaisquer máquinas, só é necessário configurar a comunicação entre os dois componentes. O provedor de computação em nuvem permite instanciar uma máquina virtual com o *master* do Kubernetes já instalado, faltando apenas configurar o *node* e a comunicação entre eles.

Figura 10 – Opções de Administrador do Harbor Registry



Fonte: Autor

Portanto, a parte descrita neste capítulo será a configuração de um *node* e sua conexão com o *master*. Para tanto, além das configurações básicas do servidor Ubuntu descritas na seção 5.1, foi necessário instalar o Docker. Isso se deve ao fato de que o *node* servirá para hospedar os *pods*, que por sua vez rodam um ou mais contêineres Docker.

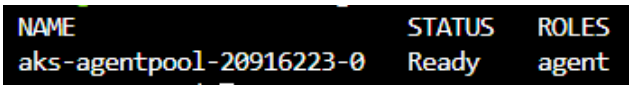
Ao instanciar o *master* numa máquina virtual em nuvem, são fornecidas credenciais - em formato de um *token* - para que o *node* consiga conectar-se a ele de uma forma segura. Portanto, para configurar o *node* são inseridas essas credenciais por meio do comando do Código 3:

Código 3 – Conexão Entre *Node* e *Master*

```
1 kubeadm join ${masterIP} \
2 --token ${masterToken} \
3 --discovery-token-ca-cert-hash ${tokenHash} \
```

Inserindo o comando `kubectl get nodes` no ambiente que hospeda o *master* é possível visualizar que a conexão foi estabelecida, pois há um novo *node*, como mostrado na Figura 11

Figura 11 – *Node* conectado ao *Master*



Fonte: Autor

5.5 GitLab CI/CD

Como citado na seção 4.5, a integração das etapas descritas neste capítulo será feita por meio de uma ferramenta denominada GitLab CI/CD. Essa ferramenta é nativamente integrada com o próprio GitLab, configurado na seção 5.2, de modo que, no instante que um *commit* é realizado as instruções escritas no arquivo `.gitlab-ci.yml` são executadas.

Para tanto, é necessário configurar *runners* que recebam as instruções e as executem. Há diversas formas de configurá-los, podendo ser diretamente no sistema operacional - Windows, Linux ou macOS - e ainda em contêineres Docker. Para o projeto foi escolhida a última opção, visto que os passos para chegar ao resultado final são mais simples.

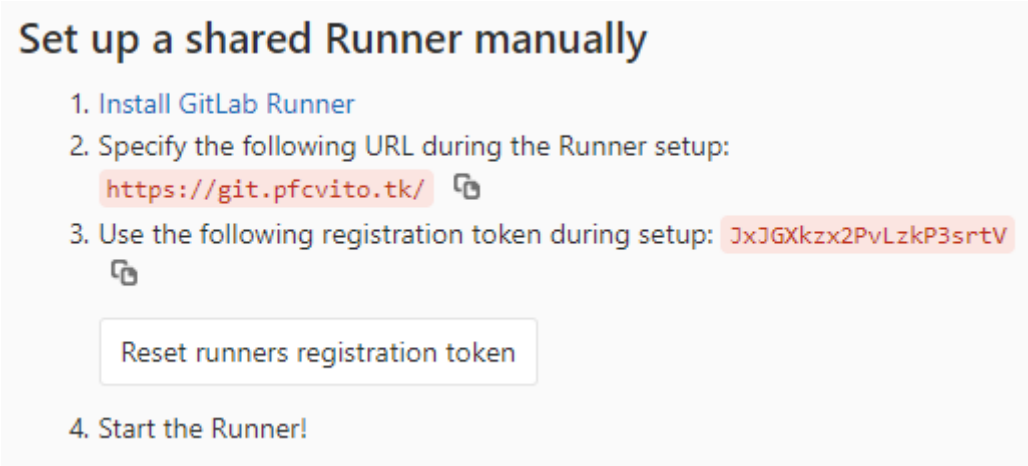
Após configurar a máquina virtual que hospedará o *runner*, como descrito na seção 5.1, foi necessário instalar o Docker. Posteriormente, como o *runner* será executado em um contêiner, foi feito o download da sua imagem a partir do registro oficial, mantido pela GitLab no DockerHub [16]. Além disso, esse contêiner foi instanciado, iniciando o funcionamento do *runner*.

Porém, o *runner* precisa estar conectado ao GitLab para receber instruções, executá-las e retornar o responder com o resultado. Para tanto, o GitLab fornece um endereço URL, juntamente com um *token* de registro, que deverão ser inseridos nas configurações do *runner* para ele conseguir iniciar a comunicação. A Figura 12 mostra essas informações na interface do GitLab.

No entanto, como é possível rodar aplicações que necessitam de dependências definidas, como um tipo específico de compilador, em uma máquina genérica como o *runner*? Isso é possível pois dentro do contêiner que está rodando o *runner* - que nada mais é que um sistema operacional Linux em um contêiner - são instanciados novos contêineres a partir de imagens que contém todas as dependências necessárias para a execução do código. Ou seja, caso uma aplicação necessite de um compilador Java será criado um contêiner, dentro do *runner*, a partir de uma imagem Docker com suporte ao *build* dessa linguagem de programação. Isso possibilita ao *runner* executar todo e qualquer código, desde que seja possível fazer o download de imagens que suportem a execução desses códigos.

O Código 4 deve ser executado, com permissão de administrador, na máquina virtual para configurar o *runner*. É importante notar que a URL e o *token* são infor-

Figura 12 – Informações para Conexão do *Runner*



Fonte: Autor

mados no momento da configuração. Além disso, o comando `gitlab/gitlab-runner` indica o nome da imagem do contêiner. Como não é indicado um registro privado, o comando `docker run` busca a imagem indicada no registro padrão, o Docker Hub. Já a *tag* `--docker-image alpine:latest` indica que deverá ser utilizada a versão `alpine`, que possui tamanho reduzido em comparação com a versão completa da imagem.

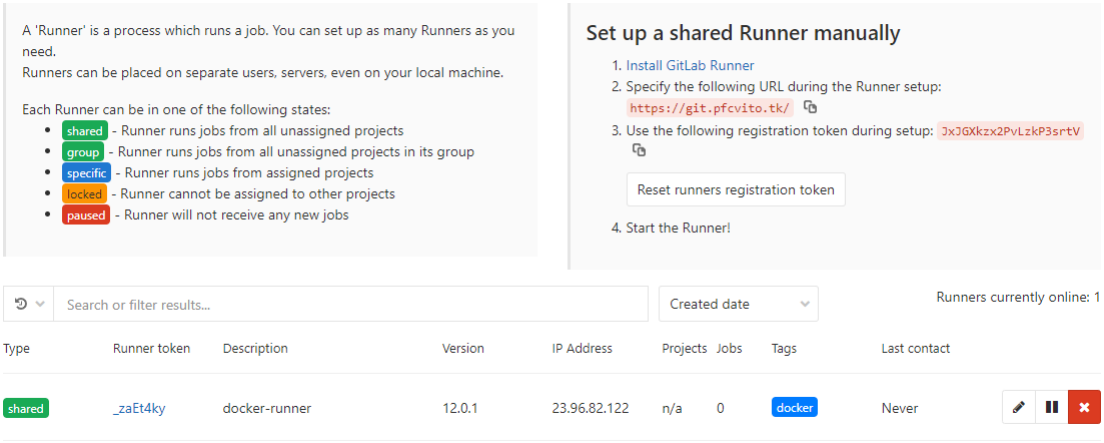
Código 4 – Criação do *Runner*

```
1  docker run --rm -v /srv/gitlab-runner/config:/etc/gitlab-runner
       \
2  gitlab/gitlab-runner register \
3  --non-interactive \
4  --executor "docker" \
5  --docker-image alpine:latest \
6  --url "https://git.pfcvito.tk/" \
7  --registration-token "JxJGXkzx2PvLzkP3srtV" \
8  --description "docker-runner" \
9  --run-untagged="true" \
10 --locked="false" \
11 --access-level="not_protected"
```

Após executar o Código 4, é possível visualizar um novo *runner* habilitado nas configurações do GitLab, conforme mostra a Figura 13. É possível configurar inúmeros *runners* e distribuí-los entre os projetos, ou seja, um *runner* só executará *jobs* dos projetos atribuídos a ele. Na etapa anterior foi configurado um *runner* compartilhado, que pode executar *jobs* de todos os projetos. Pode-se observar que, na máquina virtual que está

responsável por executar os *runners* é possível verificar se o contêiner está instanciado, como mostra a figura Figura 14

Figura 13 – *Runner* Configurado



Fonte: Autor

Figura 14 – Instância do Contêiner do *Runner*

```
root@Runners:/home/runner# docker ps
CONTAINER ID        IMAGE                                COMMAND
c26aa8cba005        gitlab/gitlab-runner:alpine-v11.2.0  "/usr/bin/dumb-init …"
```

Fonte: Autor

Para fins de monitoramento, é possível visualizar uma lista com todos os *jobs* do *runner* e seus resultados (sucesso ou falha). Isso é mostrado na Figura 15. Além disso, é possível ver a qual *commit* o *job* está associado.

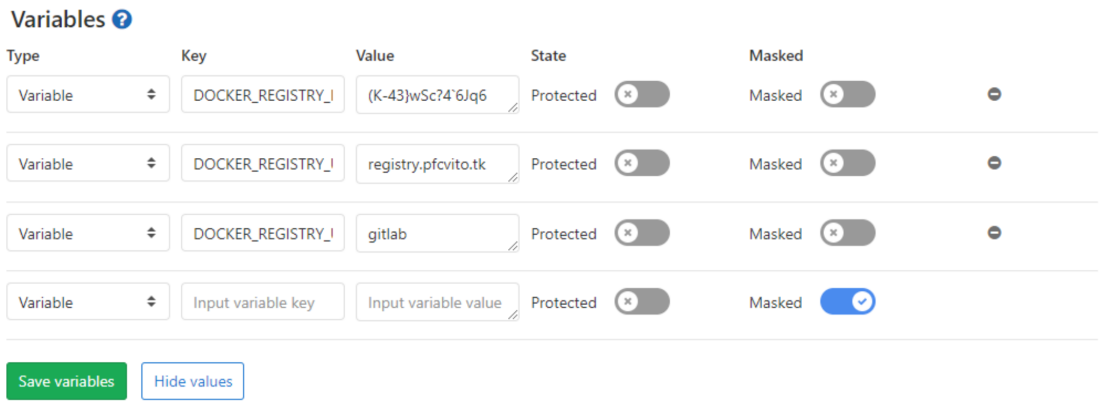
Porém, até agora, só a comunicação entre o *runner* e o GitLab foi configurada. Ainda é necessário habilitar o acesso do *runner* ao registro, configurado na seção 5.3, e ao Kubernetes, configurado na seção 5.4. Primeiramente será configurado o acesso ao registro. Para tal, o GitLab possibilita inserir no *runner* variáveis de ambiente. Elas são acessíveis somente pelo *runner* durante a execução do *job* e estão definidas diretamente no GitLab. A Figura 16 mostra a configuração dessas variáveis no ambiente do GitLab.

O acesso às variáveis de ambiente é feito por meio de sua indicação no arquivo `.gitlab-ci.yml`, como mostra o Código 5. É possível notar que o comando `docker login` realiza a autenticação no registro privado Harbor Registry, fazendo com que todos os comandos seguintes - encapsulados pela *tag script* - utilizem esse registro para as operações, como o `docker push`, que envia para o registro a imagem montada no passo `docker build`.

Figura 15 – *Jobs* do *Runner* Configurado

Fonte: Autor

Figura 16 – Variáveis de Ambiente - Harbor Registry



Fonte: Autor

A autenticação no *master* do Kubernetes também é feita através de credenciais armazenadas em variáveis de ambiente. Além disso, da mesma forma que o *build* e as demais operações com as imagens Docker, os comandos para iniciar uma nova aplicação (ou atualizá-la) no Kubernetes estão no arquivo `.gitlab-ci.yml`, porém descritos na etapa de *deploy*, mostrada no Código 6.

Porém, diferentemente do acesso a um repositório do registro - que é compartilhado

Código 5 – Etapa de *Build*

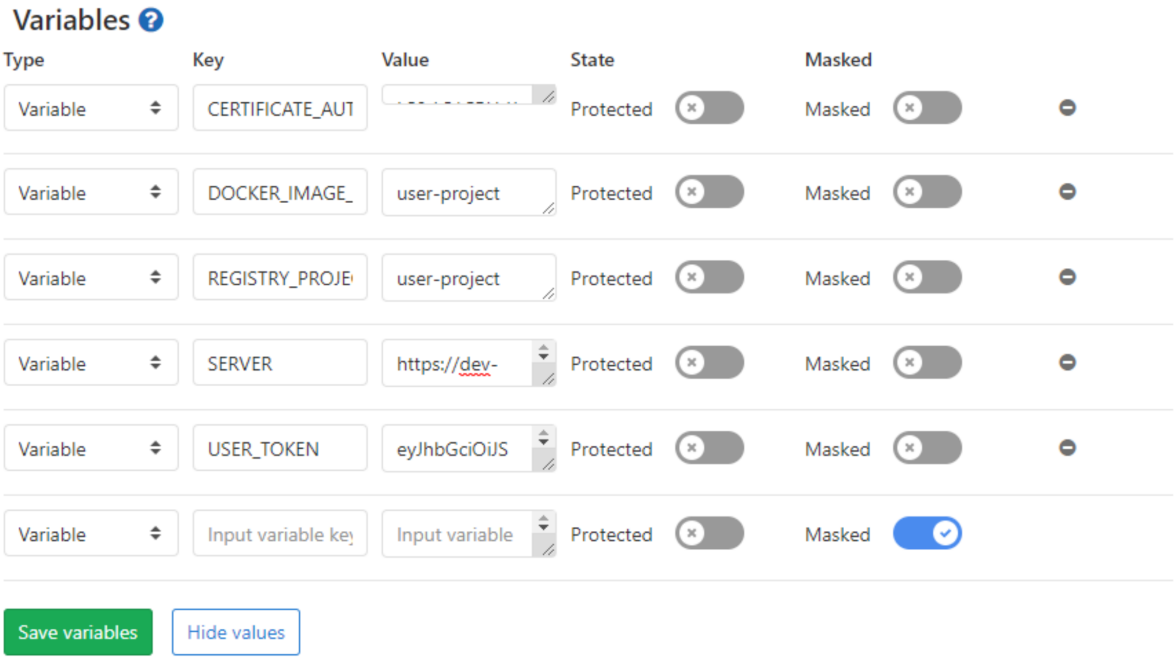
```
1  stages:
2    - build
3
4  variables:
5    CONTAINER_IMAGE: $DOCKER_REGISTRY_URL/$REGISTRY_PROJECT_NAME
6
7  build:
8    image: docker:latest
9    stage: build
10   before_script:
11     - docker login -u $DOCKER_REGISTRY_USER -p
         $DOCKER_REGISTRY_PASSWORD $DOCKER_REGISTRY_URL
12   after_script:
13     - docker logout $DOCKER_REGISTRY_URL
14   script:
15     - docker build -t $REGISTRY_PROJECT_NAME .
16     - docker tag $REGISTRY_PROJECT_NAME
17     - docker push ${CONTAINER_IMAGE}
```

Código 6 – Etapa de *Deploy*

```
1  stages:
2    - deploy
3  deploy:
4    stage: deploy
5    script:
6      - kubectl config set clusters.dev.certificate-authority-data
         ${CERTIFICATE_AUTHORITY_DATA}
7      - kubectl config set-credentials gitlab --token="${
         USER_TOKEN}"
8      - kubectl config set-context default --cluster=dev --user=
         gitlab
9      - kubectl apply -f deployment.yaml
```

entre todos os projetos de um mesmo grupo no GitLab - o acesso ao Kubernetes é segregado por projetos. Isso significa que cada projeto deve ter variáveis de ambientes, mostradas na Figura 17, que indiquem em qual *master* deverão ser executados os comandos - encapsulados pela tag *script* - descritos no arquivo `.gitlab-ci.yml`.

Figura 17 – Variáveis de Ambiente - Kubernetes



Fonte: Autor

6 Implementação: Estudo de Caso

Para melhor compreender o funcionamento dos ambientes configurados no Capítulo 5 foi elaborado um projeto que passará por todo o fluxo de desenvolvimento, denominado *pipeline*, desde o *upload* de seu código para o GitLab, construção da imagem do seu contêiner Docker e seu posterior armazenamento no Harbor Registry até o *deploy* no Kubernetes. Portanto, o foco deste capitulo não é a aplicação em si, mas sim o percurso que ela percorrerá nas ferramentas configuradas.

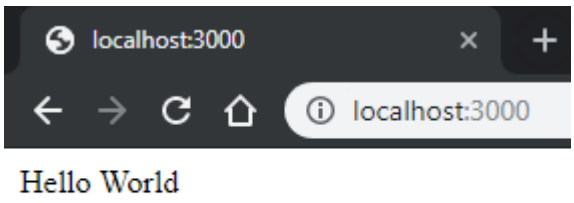
A aplicação para demostração foi elaborada em Node.js - um interpretador de código JavaScript que possibilita criar aplicações - e é basicamente um servidor *web* escutanto requisições `get` na porta `3000` e respondendo com o *status* `200` juntamente com um texto. O Código 7 mostra essa aplicação exemplo, que na linha 6 define o tipo de resposta para as requisições `get` e na linha 8 escuta conexões na porta `3000`.

Código 7 – Aplicação Exemplo

```
1 const express = require('express');
2 const app = express();
3
4 app.route('/')
5     .get((req, res) => {
6         res.status(200).send("Hello World");
7     });
8 app.listen(3000);
```

Executando o exemplo acima localmente é possível acessá-lo através do endereço `localhost:3000`, como mostra a Figura 18.

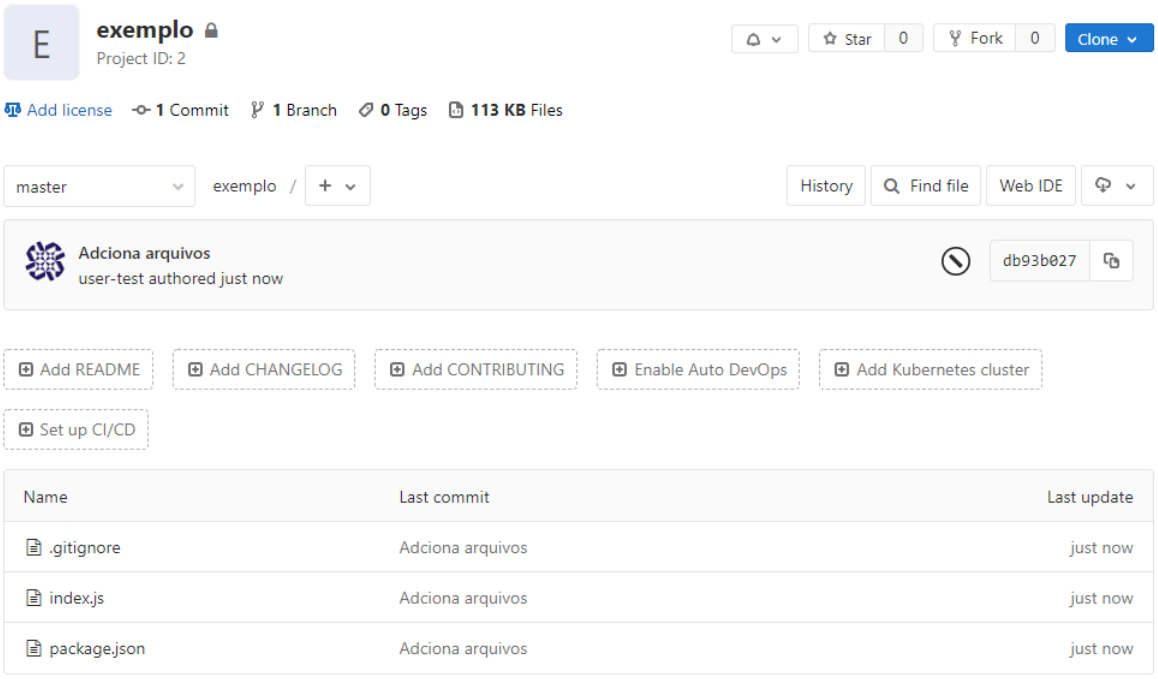
Figura 18 – Exemplo Local



Fonte: Autor

Após essa etapa, é necessário criar um projeto no GitLab, fazer o *clone* para a máquina local - através do comando `git clone` - colocar o código fonte na pasta do projeto e efetuar um *commit*. O resultado desse processo é mostrado na Figura 19. Além do arquivo do código fonte há dois arquivos, o `.gitignore` - que indica quais arquivos não deverão ser versionados pelo GitLab - e o `package.json`, que contém a declaração das dependências do projeto.

Figura 19 – Projeto no GitLab



Fonte: Autor

Esse *commit* não disparou nenhum *pipeline*, visto que não há um arquivo, entitulado `.gitlab-ci.yaml`, indicando os comandos necessários. Portanto, o próximo passo é adicionar esse arquivo, mostrado no Código 8.

Código 8 – .gitlab-ci.yml

```
1 stages:
2   - build
3   - deploy
4
5 variables:
6   CONTAINER_IMAGE_NAME: $DOCKER_REGISTRY_URL/
        $REGISTRY_PROJECT_NAME/$DOCKER_IMAGE_NAME: ${
        CI_COMMIT_REF_SLUG}-${CI_COMMIT_SHA}
7
```

```
 8 services:
 9   - docker:dind
10
11 build:
12   image: docker:latest
13   stage: build
14   before_script:
15     - docker login -u $DOCKER_REGISTRY_USER -p
         $DOCKER_REGISTRY_PASSWORD $DOCKER_REGISTRY_URL
16   after_script:
17     - docker logout $DOCKER_REGISTRY_URL
18   script:
19     - docker build -t $REGISTRY_PROJECT_NAME:${
         CI_COMMIT_REF_SLUG}.${CI_COMMIT_SHA:0:8} .
20     - docker tag $REGISTRY_PROJECT_NAME:${CI_COMMIT_REF_SLUG}.${
         CI_COMMIT_SHA:0:8} ${CONTAINER_IMAGE_NAME}
21     - docker push ${CONTAINER_IMAGE_NAME}
22
23 deploy:
24   stage: deploy
25   image: alpine
26   script:
27     - apk add --no-cache curl
28     - curl -LO https://storage.googleapis.com/kubernetes-release
         /release/$(curl -s https://storage.googleapis.com/
         kubernetes-release/release/stable.txt) /bin/linux/amd64/
         kubectl
29     - chmod +x ./kubectl
30     - mv ./kubectl /usr/local/bin/kubectl
31
32     - kubectl config set-cluster dev --server="${SERVER}"
33     - kubectl config set clusters.dev.certificate-authority-data
         ${CERTIFICATE_AUTHORITY_DATA}
34     - kubectl config set-credentials gitlab --token="${
         USER_TOKEN}"
35     - kubectl config set-context default --cluster=dev --user=
         gitlab
36     - kubectl config use-context default
37     - sed -i "s <CONTAINER_IMAGE_NAME> ${CONTAINER_IMAGE_NAME} g
```

```
      " deployment.yaml;
38    - kubectl apply -f deployment.yaml
```

O *pipeline* é dividido em dois estágios: *build* e *deploy*, indicados nas linhas (2) e (3). No primeiro a imagem Docker é montada e adicionada ao Harbor Registry. Já no segundo, a imagem recém construída é instanciada em um *pod* do Kubernetes e disponibilizada para acesso através do endereço do *node*, configurado pelo `kube-proxy`, conforme descrito na subseção 4.4.2.

Um passo importante está na linha (6), onde é definida uma variável para armazenar o nome da imagem do contêiner. Os três primeiros parâmetros são variáveis de ambiente, que ajudam a classificar e segregar as imagens por projeto no Harbor Registry. Já os últimos dois parâmetros são necessários para indicar a diferença entre as imagens do mesmo projeto. A variável `${CI_COMMIT_REF_SLUG}` indica o nome da *branch* na qual foi realizado o *commit* que disparou o *pipeline*. Já a variável `${CI_COMMIT_SHA}` é um indicador único para cada *commit*, gerado pelo GitLab.

Na linha (11) começa a descrição do estágio de *build*. Primeiramente, é necessário realizar a autenticação do *runner* no registro, a fim de possibilitar o *upload* de imagens Docker. Isso é feito na linha (15) com o comando `docker login`. Posteriormente, na linha (17), é indicado o *script* de desconexão com o registro, através do comando `docker logout`, após o fim de todos os comandos do estágio.

A construção da imagem Docker é feita pelo comando da linha (19), o qual executa as instruções do `Dockerfile` e salva o resultado em uma imagem. Porém, para o sucesso dessa etapa necessário o arquivo `Dockerfile`, que é mostrado no Código 9. Na linha (4) deste arquivo é feito o *download* de todas as dependências indicadas no `package.json`. Já na linha (5) é indicado o comando necessário para iniciar a aplicação contida pelo contêiner.

Código 9 – Dockerfile

```
1 FROM node:alpine
2 COPY . .
3 WORKDIR .
4 RUN npm install
5 CMD ["npm", "run", "start"]
```

Voltando ao Código 8, após a conclusão da construção da imagem Docker é necessário enviá-lá para o Harbor Registry, conforme mostra a linha (21). Após o sucesso dessa etapa a próxima é automaticamente iniciada.

A etapa de *deploy* é responsável por instanciar um novo contêiner no *node* com a imagem construída na etapa *build*, a partir de um arquivo denominado `deployment.yaml`.

Esse arquivo será descrito posteriormente. Os comandos das linhas (27) à (30) do Código 8 fazem o *download* e a configuração do `kubectl` para possibilitar a comunicação com o *cluster* do Kubernetes. Nas linhas (32) à (36) é configurado o acesso do `kubectl` ao *cluster*, possibilitando enviar comandos através do *runner*. Na linha (37) é feita a substituição do nome da imagem, no arquivo `deployment.yaml`, pela imagem montada na etapa *build*. Isso é necessário para que os contêineres sejam atualizados com as versões mais recentes.

Código 10 – deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: hello-express
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: hello-express
10   template:
11     metadata:
12       labels:
13         app: hello-express
14     spec:
15       containers:
16       - name: hello-express
17         image: <CONTAINER_IMAGE_NAME>
18         resources:
19           requests:
20             cpu: 100m
21             memory: 128Mi
22           limits:
23             cpu: 250m
24             memory: 256Mi
25         ports:
26         - containerPort: 3000
27           name: http
28           protocol: TCP
29       imagePullSecrets:
30       - name: registry-secret
31 ---
32 apiVersion: v1
```

```
33 kind: Service
34 metadata:
35   name: hello-express
36 spec:
37   type: LoadBalancer
38   ports:
39   - port: 3000
40   selector:
41     app: hello-express
```

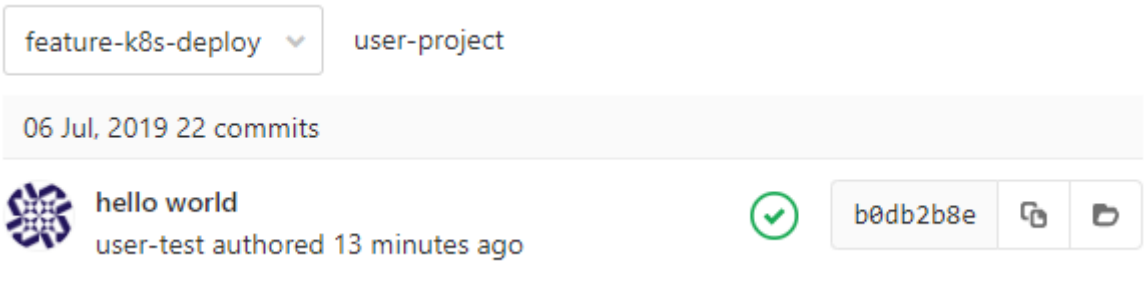
O Código 10 apresenta o arquivo que descreve como deverá ser feito o *deploy* da aplicação. A linha (6) indica que é desejado apenas um contêiner da aplicação, nomeada na linha (9) pela *tag* `app`. A partir da linha (14), até a (30), são definidas as especificações para o *deploy*. Na linha (17) existe a *tag* `image`, na qual é definido o nome da imagem que deverá ser utilizada para instanciar o contêiner. Ela está com uma variável, que será substituída pelo nome da imagem através do comando da linha (37) do Código 8.

Posteriormente, da linha (18) à (24) são definidos o mínimo necessário e o limite de recursos para o contêiner. Nas linhas (25) e (26) é indicada a porta que a aplicação utiliza para receber requisições, de acordo com o Código 7.

Porém, além de indicar como a aplicação deve ser instanciada, é necessário indicar como ela deve ser disponibilizada. Isso é feito nas linhas (32) à (41), que cria um serviço para realizar essa tarefa.

Após adicionar esses arquivos no diretório do projeto, é possível realizar um *commit* e verificar o comportamento das ferramentas configuradas. A Figura 20 mostra o *commit* com o identificador associado `b0db2b8e`, que será utilizado nas próximas etapas.

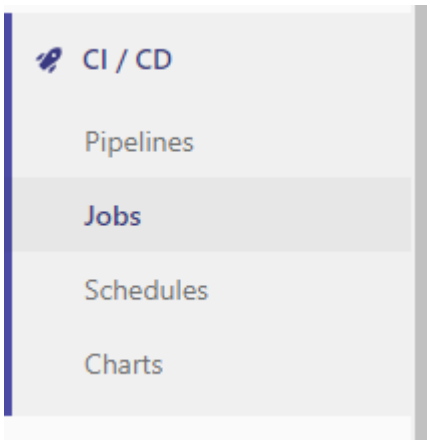
Figura 20 – *Commit*



Fonte: Autor

O GitLab possui uma área específica para acompanhar os *pipelines*, conforme mostra a Figura 21. Nesta seção é possível visualizar todos as etapas do *pipeline*.

Figura 21 – Opções de Monitoramento de *pipelines* no GitLab



Fonte: Autor

O pipeline é disparado assim que o *commit* é concluído, conforme mostra a Figura 22. É possível notar que o *build* associado ao *commit* está concluído com sucesso e o *deploy* está sendo executado.

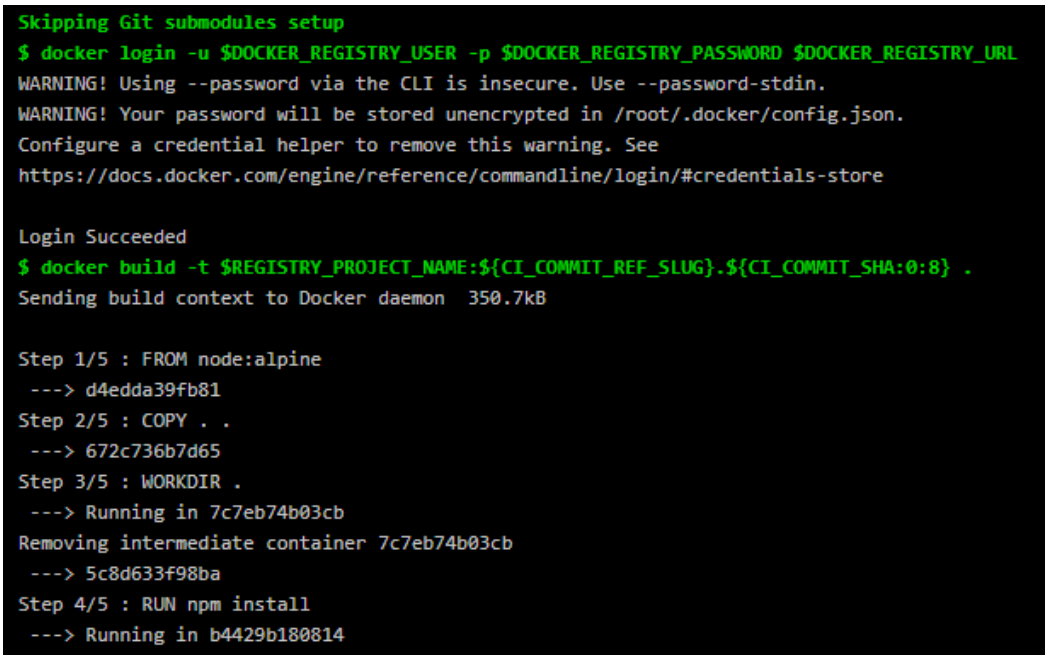
Figura 22 – *Jobs* Executados

Fonte: Autor

Além disso, é possível acompanhar em tempo real o que está sendo executado pelo *runner*, conferindo inclusive os resultados de cada comando descrito no `.gitlab-ci.yml`.

A etapa de *build* está dividida em duas figuras, a primeira na Figura 23 mostra os comandos de autenticação no registro Harbor Registry - utilizando as variáveis de ambientes mostradas na Figura 16 - e de *build* da imagem do contêiner, utilizando o Código 9. Já a Figura 24 mostra os comandos de classificação da imagem Docker, através da inserção de uma *tag*, do *upload* da imagem para o registro e do *logout* do registro, além da indicação de sucesso da etapa de *build*.

Figura 23 – Instruções de *Build* no *Runner* - Parte 1



Fonte: Autor

Como há indicação, nos comandos do *runner*, que a imagem Docker foi enviada para o registro, é possível conferir isso por meio do acesso ao Harbor Registry e verificação da presença de uma imagem que contenha no nome o identificador do *commit*. Esse passo é mostrado na Figura 25.

Conforme indicado no `.gitlab-ci.yml`, através da *tag stages*, a etapa seguinte à *build* é o *deploy*. Assim como no *build*, essa etapa é dividida em duas imagens. A Figura 26 mostra a primeira parte dos comandos executados no *runner*. É possível acompanhar o *download* do `kubectl` e sua posterior configuração para acessar o *cluster* do Kubernetes. As credenciais utilizadas nessa etapa também estão armazenadas em variáveis de ambiente do GitLab, que são transferidas ao *runner*. Já a Figura 27 mostra a alteração, no arquivo `deployment.yaml`, do nome da imagem. O comando `sed` substitui a variável `<CONTAINER_IMAGE_NAME>` pelo nome da imagem construída na etapa de *build*. Posteriormente, com o comando `kubectl apply`, as configurações do arquivo `deployment.yaml` são aplicadas.

Após a conclusão das etapas de *build* e *deploy* é possível visualizar o sucesso do *commit*, mostrado na imagem Figura 28.

Posteriormente, acessando o *master* no Kubernetes, pode ser visualizado o *deployment* realizado nas etapas anteriores através do comando `kubectl get deployments`, tendo seu resultado mostrado na Figura 29. Além disso, como também é configurado um *Service* responsável por disponibilizar o *deployment* externamente para a Internet,

Figura 24 – Instruções de *Build* no *Runner* - Parte 2

```
Successfully built 04398a799a86
Successfully tagged user-project:feature-k8s-deploy.b0db2b8e
$ docker tag $REGISTRY_PROJECT_NAME:${CI_COMMIT_REF_SLUG}.${CI_COMMIT_SHA:0:8} ${CONTAINER_IMAGE_NAME}
$ docker push ${CONTAINER_IMAGE_NAME}
The push refers to repository [registry.pfcvito.tk/user-project/user-project]
c80f2a78847d: Preparing
7c3fd2a6663d: Preparing
731bc48ec47f: Preparing
67034be71e6d: Preparing
3f896cfe4663: Preparing
f1b5933fe4b5: Preparing
f1b5933fe4b5: Waiting
67034be71e6d: Layer already exists
3f896cfe4663: Layer already exists
731bc48ec47f: Layer already exists
f1b5933fe4b5: Layer already exists
7c3fd2a6663d: Pushed
c80f2a78847d: Pushed
feature-k8s-deploy-b0db2b8e65fbc5bd28adbba3fbaf93e2f55b0de1: digest:
sha256:56fe0bec5aca1e2c805420593ac4237ff34e0c0b26d68f696b5f12f961fb70da size: 1578
Running after script...
$ docker logout $DOCKER_REGISTRY_URL
Removing login credentials for registry.pfcvito.tk
Job succeeded
```

Fonte: Autor

Figura 25 – Imagem no Registro

Fonte: Autor

consegue-se conferir o resultado através do comando `kubectl get services`. A Figura 30 mostra as informações do *service*, indicando principalmente o endereço de IP que possibilita o acesso à aplicação.

Como os contêineres estão instanciados em um *pod*, é também possível verificar se os *pods* foram criados ou atualizados através do comando `kubectl get pods`, com o resultado mostrado na Figura 31.

Figura 26 – Instruções de *Deploy* no *Runner* - Parte 1

```
$ apk add --no-cache curl
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/community/x86_64/APKINDEX.tar.gz
(1/4) Installing ca-certificates (20190108-r0)
(2/4) Installing nghttp2-libs (1.38.0-r0)
(3/4) Installing libcurl (7.65.1-r0)
(4/4) Installing curl (7.65.1-r0)
Executing busybox-1.30.1-r2.trigger
Executing ca-certificates-20190108-r0.trigger
OK: 7 MiB in 18 packages
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 40.9M  100 40.9M    0     0  63.1M      0 --:--:-- --:--:-- --:--:-- 63.1M
$ chmod +x ./kubectl
$ mv ./kubectl /usr/local/bin/kubectl
$ kubectl config set-cluster dev --server="${SERVER}"
Cluster "dev" set.
$ kubectl config set clusters.dev.certificate-authority-data ${CERTIFICATE_AUTHORITY_DATA}
Property "clusters.dev.certificate-authority-data" set.
$ kubectl config set-credentials gitlab --token="${USER_TOKEN}"
User "gitlab" set.
$ kubectl config set-context default --cluster=dev --user=gitlab
Context "default" created.
$ kubectl config use-context default
Switched to context "default".
```

Fonte: Autor

Figura 27 – Instruções de *Deploy* no *Runner* - Parte 2

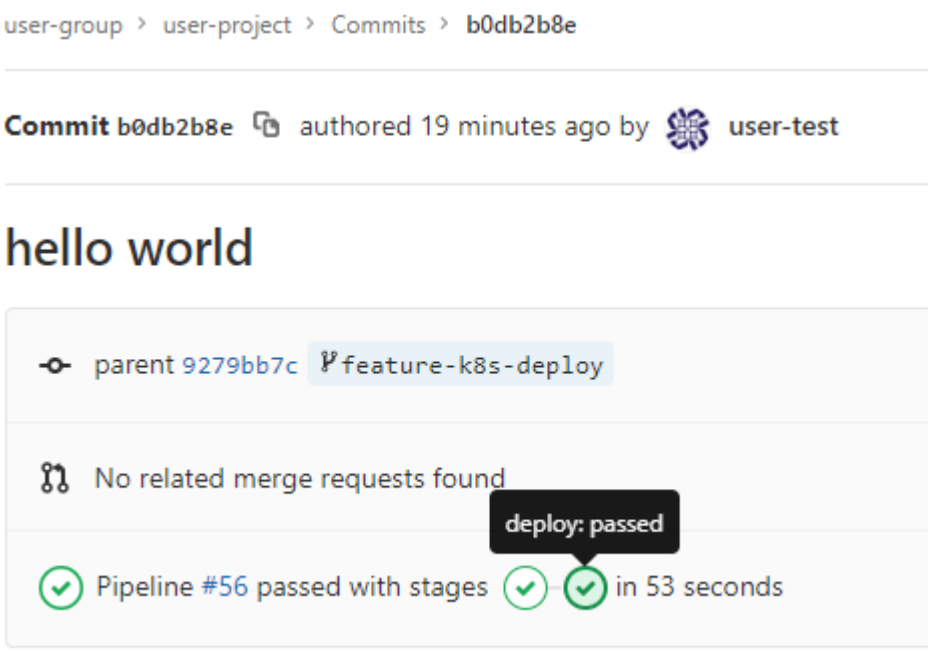
```
$ sed -i "s <CONTAINER_IMAGE_NAME> ${CONTAINER_IMAGE_NAME} g" deployment.yaml;
$ kubectl apply -f deployment.yaml
deployment.apps/hello-express configured
service/hello-express unchanged
Job succeeded
```

Fonte: Autor

Além disso, para conferir se a imagem do contêiner criada a partir do *commit* corresponde à imagem que foi utilizada para instanciar o contêiner do *pod*, é possível utilizar o comando `kubectl describe pods`, que mostra as especificações do *pod*, inclusive a imagem associada ao *commit*. Esse resultado é mostrado na Figura 32, que condiz com o esperado pois a *tag* `Image` mostra a versão da imagem com o identificado do *commit* realizado.

Acessando o endereço de IP do serviço através de um navegador, obtêm-se o

Figura 28 – Indicação de Sucesso do *Commit*



Fonte: Autor

Figura 29 – Informações do *Deployment*

```
NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
hello-express   1         1         1            1
```

Fonte: Autor

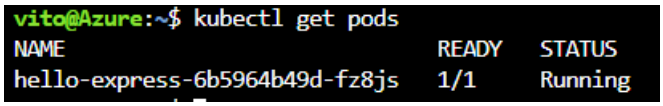
Figura 30 – Informações do *Service*

```
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
hello-express   LoadBalancer   10.0.151.64    40.71.177.240   3000:31993/TCP
```

Fonte: Autor

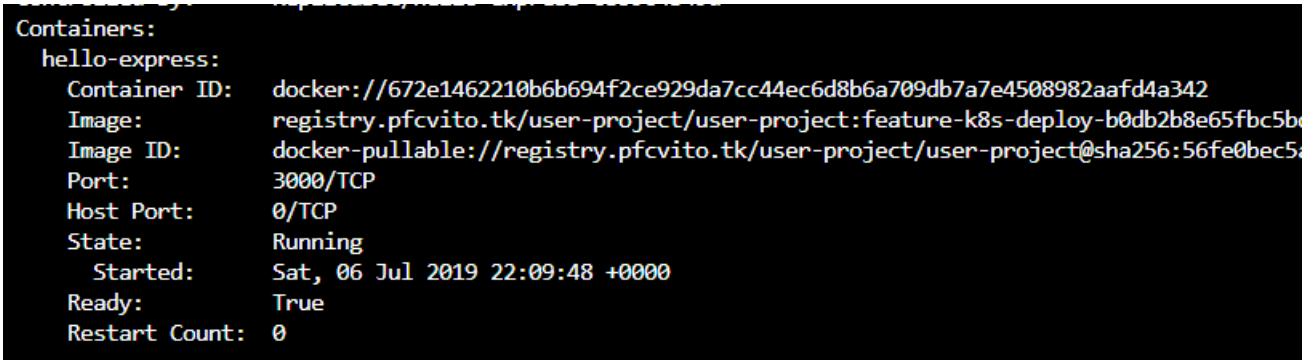
resultado final do projeto, que é a aplicação disponível para o cliente. Isso é mostrado na Figura 33.

Figura 31 – Informações do *Pod*



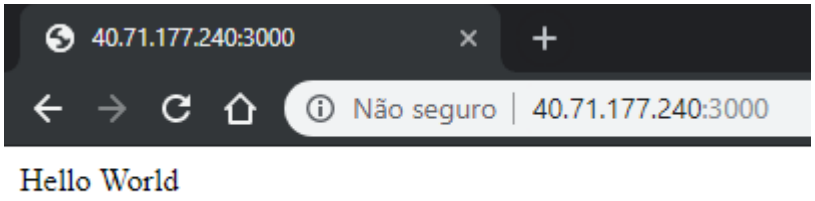
Fonte: Autor

Figura 32 – Descrição do *Pod*



Fonte: Autor

Figura 33 – Acesso à Aplicação



Fonte: Autor

7 Conclusão

Inicialmente foi proposto a automação de um processo manual e trabalhoso: a implantação de sistemas. Esse objetivo foi definido pois foi identificado um gargalo na entrega dos softwares, localizado justamente nessa etapa manual. Posteriormente, após classificar as atividades suscetíveis à automação, foram pesquisadas soluções que atendessem às necessidades do projeto. Portando, foi definido que seriam utilizadas ferramentas completas, sendo necessário somente configurá-las.

Para tanto, fez-se uso de vários ambientes para hospedar essas soluções, visto que era indispensável que elas fossem acessíveis por todos os colaboradores da empresa. Posteriormente, foram configuradas as ferramentas, ajustando cada uma delas para atender às especificações definidas. Porém, a automação não é efetiva caso essas ferramentas não sejam capazes de receber dados, efetuar um processamento e transferi-los para a próxima etapa. Portanto, todas as ferramentas foram integradas a fim de criar um fluxo único e automatizado dos processos.

Para atestar a eficácia do ambiente configurado, foi mostrado um exemplo que, a partir do seu código fonte, percorreu todas as ferramentas através do fluxo automático definido, finalizando na sua implantação em um servidor.

Frente aos objetivos propostos, pode-se afirmar que o projeto atingiu os objetivos, fato verificado no estudo de caso apresentado, que mostra uma aplicação sendo implementada automaticamente a partir de seu código fonte, sendo esse justamente o principal objetivo do projeto. Ademais, os objetivos específicos - que são passos que possibilitam atingir o objetivo geral de automatizar a implantação de softwares - também foram cumpridos, visto que todas as etapas elencadas foram executadas.

Referências

1 PRIES, K. H.; QUIGLEY, J. M. *Scrum Project Management*. Boca Raton, Flórida, Estados Unidos: CRC Press, 2011. Citado 2 vezes nas páginas 25 e 26.

2 SCHWABER, K.; SUTHERLAND, J. *The Scrum Guide*. [S.l.: s.n.], 2017. Citado na página 26.

3 SOMASUNDARAM, R. *Git: Version Control for Everyone*. Birmingham, Reino Unido: Packt Publishing, 2013. Citado na página 27.

4 CHACON, S. *Pro Git*. New York, NY, Estados Unidos: Apress, 2009. Citado na página 27.

5 BARRETT, D. J.; SILVERMAN, R. *SSH, The Secure Shell: The Definitive Guide*. Sebastopol, CA, Estados Unidos: O'Reilly, 2001. Citado na página 28.

6 MICROSOFT. *Introdução aos contêineres e ao Docker*. <https://docs.microsoft.com/pt-br/dotnet/standard/microservices-architecture/container-docker-introduction/>. Acesso em: 19/06/2019. Citado na página 28.

7 SAITO, H.; CHLOE LEE, H. C.; WU, C. Y. *DevOps with Kubernetes*. Birmingham, Reino Unido: Packt Publishing, 2017. Citado 3 vezes nas páginas 28, 29 e 30.

8 WAHABALLA, A. et al. Toward unified devops model. *2015 6th IEEE International Conference on Software Engineering and Service Science*, 2015. Citado na página 30.

9 VIRMANI, M. Understanding devops and bridging the gap from continuous integration to continuous delivery. *Fifth international conference on Innovative Computing Technology*, 2015. Citado na página 30.

10 KUBERNETES. *Kubernetes Documentation*. <https://kubernetes.io/>. Acesso em: 22/06/2019. Citado na página 37.

11 OPENSSH. *OpenSSH*. <http://www.openssh.com/>. Acesso em: 19/06/2019. Citado na página 45.

12 UBUNTU. *Uncomplicated Firewall*. <https://help.ubuntu.com/community/UFW>. Acesso em: 19/06/2019. Citado na página 45.

13 LET'S ENCRYPT. *Documentation*. <https://letsencrypt.org/>. Acesso em: 19/06/2019. Citado na página 45.

14 GITLAB. *GitLab Documentation*. <https://about.gitlab.com/>. Acesso em: 22/06/2019. Citado na página 46.

15 HARBOR REGISTRY. *User documents*. <https://goharbor.io/>. Acesso em: 22/06/2019. Citado na página 48.

16 DOCKER HUB. *Docker Hub Quickstart*. <https://hub.docker.com/>. Acesso em: 28/06/2019. Citado na página 53.