UNIVERSIDADE EVANGÉLICA DE ANÁPOLIS – UNIEVANGÉLICA
ENGENHARIA DE SOFTWARE

GUSTAVO DOS SANTOS SOBRINHO
RAPHAELA PAOLLA DOS SANTOS

Estruturação de processo para desenvolvimento utilizando DevOps

ANÁPOLIS - GO
Novembro, 2021

UNIVERSIDADE EVANGÉLICA DE ANÁPOLIS – UNIEVANGÉLICA
ENGENHARIA DE SOFTWARE

GUSTAVO DOS SANTOS SOBRINHO
RAPHAELA PAOLLA DOS SANTOS

Estruturação de processo para desenvolvimento utilizando DevOps

Trabalho apresentado ao Curso de Engenharia de Software da Universidade Evangélica de Goiás – UniEVANGÉLICA, da cidade de Anápolis-GO como requisito parcial para obtenção do Grau de Bacharel em Engenharia de Software.

Orientador(a): Prof. Millys Fabrielle Araujo Carvalhaes.

Coorientador(a): Prof. Eduardo Ferreira de Souza

ANÁPOLIS - GO
Novembro, 2021

UNIVERSIDADE EVANGÉLICA DE ANÁPOLIS – UNIEVANGÉLICA
ENGENHARIA DE SOFTWARE

GUSTAVO DOS SANTOS SOBRINHO
RAPHAELA PAOLLA DOS SANTOS

Estruturação de processo para desenvolvimento utilizando DevOps

Monografia apresentada para Trabalho de Conclusão de Curso de Engenharia de Software da Universidade Evangélica de Goiás - UniEVANGÉLICA, da cidade de Anápolis-GO como requisito parcial para obtenção do grau de Engenheiro(a) de Software.

Aprovado por:

_____

(ORIENTADOR)

_____

(COORIENTADOR)

_____

(AVALIADOR)

Anápolis, 21 de novembro de 2021.

RESUMO

A atual Indústria de Software possui a necessidade de entregar funcionalidades e melhorias de seus sistemas cada vez mais rápido e com maior qualidade do produto, este fato se intensifica quando olhamos o atual crescimento da indústria atrelado ao aumento da rotatividade de funcionários, tornando o ambiente instável. Assim, objetivou-se estruturar um modelo de processo DevOps, utilizando um ambiente mínimo de desenvolvimento de software, automatizando parte do processo de deploy para uma melhoria na cadência de entregas. Através da análise dos processos de desenvolvimento das empresas e utilizando as práticas de DevOps que tratam principalmente a comunicação, colaboração, automação e monitoramento será estruturado um processo que crie um ambiente mínimo integrando as diversas práticas que fundamentam o desenvolvimento e integração contínua contidos nas práticas da metodologia em questão.

Palavras chave: Indústria de Software. Qualidade do produto. DevOps. Desenvolvimento de software. Integração.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

FS	Fábrica de Software
XP	Extreme Programing
IAC	*Infrastructure As Code*
TI	Tecnologia da Informação
E/S	Entrada e Saída
CI	*Continuous Integration*
CD	*Continuous Deployment*

INTRODUÇÃO

A Indústria de Software sofre diversas mudanças com o decorrer do tempo, buscando sempre a melhoria de seus produtos e processos de acordo com a necessidade do seu cliente, tanto no desenvolvimento quanto na manutenção, visando manter a melhor qualidade nos serviços (GUERRERO, 2020). Em meio a este ambiente, a Indústria de Software nota a necessidade de incrementos para a melhoria de sua produtividade, como o uso de ferramentas e *frameworks* para dar suporte aos processos contidos no desenvolvimento do software, visando adquirir os resultados esperados com alta qualidade em um ambiente de desenvolvimento contínuo de serviços (VIRMANI, 2015).

Desenvolvedores são cobrados por entregas de funcionalidades em aplicações que geram valor, enquanto a equipe de operações é cobrada pela estabilidade do sistema, assim ambas equipes necessitam de uma maior interação melhorando a agilidade na implementação e em inovações contínuas. Assim, indústrias com um processo que facilite a comunicação e permita colaboradores e líderes saberem como e onde agir, além de facilmente compreenderem o funcionamento do processo como um todo alcançam uma maior agilidade na entrega de funcionalidades (CRUZ, 2019).

A ausência desse processo de integração pode gerar retrabalhos para os desenvolvedores e atrasos para a equipe de operações, fazendo com que a qualidade do produto caia e haja falhas na sua agilidade em decorrência da destoante cultura e práticas organizacionais vividas por cada equipe. Assim a aplicação da cultura DevOps procura unir tanto a equipe de desenvolvimento quanto da equipe de operações unificando o cenário de atuação vivido por ambas, trazendo uma integração que visa melhorar a efetividade do desenvolvimento de funcionalidade e correções do sistema, integrando-as como um objetivo comum a todos (BARBOSA, 2019).

Porém, ainda é uma tarefa desafiadora sua implementação, por conter uma infinidade de informações, práticas e ferramentas relacionadas, de modo que ainda não está claro como essa rica, porém ainda dispersa, quantidade de informação é organizada por não conter um guia ou framework estruturado.

Foi com este objetivo subjacente que se conduziu o trabalho de investigação apresentado neste relatório, um estudo de caso sobre as práticas propostas na cultura em questão e sua aplicabilidade. Através do caso de estudo foi possível verificar o “antes” e “depois" da adoção da cultura DevOps, e estudar a sua adoção em diferentes vertentes.

Como principais resultados identificam-se a revisão da literatura efetuada e a caracterização vários aspetos como, por exemplo, as práticas (que os entrevistados identificaram como mais relevantes a continuous integration e continuous deployment), os benefícios, barreiras, (ex: resistência à mudança). Vale destacar, ainda, como principal fator influenciador do sucesso na adoção/implementação de DevOps, à semelhança de outras iniciativas nas organizações e o apoio da gestão de topo (SOUSA, 2019).

1. FUNDAMENTAÇÃO TEÓRICA

1.1 O que é e como funciona uma Fábrica de software?

O termo Fábrica de Software (FS) remete a utilização de conceitos da indústria geral em um ambiente de produção. Um processo formado por passos de subprocessos parcialmente ordenados, definindo a ideia de aplicar conceitos da indústria em geral em ambientes de desenvolvimento de *software*, de forma a aumentar a produtividade e diminuir prazos e custos, tornando o processo mais independente do fator humano. (AGUILAR, 2011)

Uma FS realiza a criação de produtos de acordo com a necessidade do seu cliente, fazendo uso de processos e operações que façam garantir a qualidade, adequando a um ciclo de desenvolvimento e produtividade (FERNANDES, 2004).

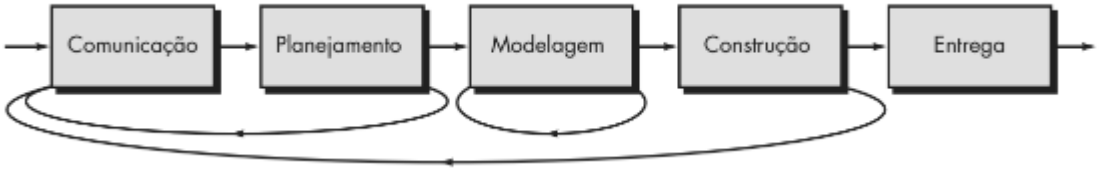
Para um bom funcionamento, nota-se a necessidade de dividir a fábrica em setores: atendimento de clientes, negociação e especificação de necessidades da área usuário, planejamento e controle de produção, definição de objetos a serem utilizados ou desenvolvidos e a equipe de garantia que verifica se o produto final atende a todas as especificações. Cada processo tem como foco um cenário diferente de produção e produto, sendo assim prioriza diferentes artefatos com base em sua metodologia. Cada etapa, da metodologia adotada, devolve parte do desenvolvimento do software em si, desde sua concepção inicial até o produto final, sendo cada etapa continuação da etapa anterior (AMADEU, GONÇALVES & TEIXEIRA JUNIOR, 2013).

1.2 Processo de software

Um processo de software pode ser entendido como uma sequência estruturada de atividades a serem realizadas durante sua produção. Um processo é um conjunto de etapas que envolvem atividades, restrições e recursos para alcançar uma saída desejada. Sendo assim, é visto como atividades e resultados que quando interligados, geram um produto de software. (SOMMERVILLE, 2004).

Segundo Pressman (2011), um processo genérico estabelece cinco atividades metodológicas: comunicação, planejamento, modelagem, construção e entrega (Figura 1).

Figura 1: Modelo de processo genérico



Fonte: PRESSMAN, 2011

Além disso, há mais um conjunto de atividades de apoio às atividades guarda-chuva que podem ser utilizadas, elas são aplicadas ao longo do projeto e ajudam a administrar riscos e garantir a qualidade. Um fluxo de processo descreve como são organizadas essas cinco atividades metodológicas.

1.2.1 MODELO DE PROCESSO INCREMENTAL

Na visão de Pressman (2011) O modelo incremental de desenvolvimento de software mistura elementos dos fluxos de processo lineares e paralelos. ele aplica uma sequência linear de produção, de forma escalonada, sendo assim à medida que o tempo avança ele gera um novo incremento como em um fluxo de processos evolucionários.

O processo incremental de desenvolvimento pode ser visto como a ideia de “aumentar um pouco” o âmbito do sistema. Uma mansão que foi construída a partir de uma casa com apenas algumas divisões utilizando incrementos cíclicos devidamente priorizados, é um bom exemplo de como o processo em questão funciona Ramos (2006). Para que esse modelo de desenvolvimento seja viável é preciso que a equipe responsável possua suas cinco atividades metodológicas bem estruturadas, com foco principal na comunicação e planejamento (RAMOS, 2006).

1.2.2 MVP

MVP é abreviação do conceito *minimum viable product* que, em português, representa o produto mínimo viável. São apenas três letras que carregam um significado muito importante para empreender com sucesso. O MVP é a versão mais simples de um produto que será criada e disponibilizada aos usuários para validar uma ideia e coletar dados imprescindíveis para validar o direcionamento do negócio (CAROLI, 2021). Em outras palavras, o MVP serve como

um motor de crescimento para projetos. Ele garante ao negócio um crescimento sustentável, atuando em conjunto com diversos outros fatores (CONAPP, 2020). É importante ressaltar que: para entregar o mínimo viável, é necessário deixar muitas funcionalidades para momentos posteriores. Sendo assim, a primeira etapa é compreender e priorizar o que compõe esse produto mínimo viável, de modo que não comprometa a qualidade do que será entregue ao usuário (CAROLI, 2021).

1.3 DevOps

O DevOps é frequentemente referido pelos profissionais da área das tecnologias da informação como um movimento cultural ou profissional, que apresenta uma nova abordagem de entrega de aplicações informáticas (*software*), através da colaboração entre as equipes de desenvolvimento e das equipes de operações. Tem subjacente um conjunto diversificado de princípios, relacionados com cultura, automatização, lean, monitorização e partilha; e práticas, tais como, *continuous integration* (CI) e *continuous deployment* (CD) (SOUSA, 2019).

- De acordo com SATO (2014) *continuous integration* (Integração contínua) é uma das práticas originais de *Extreme Programming* (XP), incentivando desenvolvedores a frequentemente integrar seu trabalho para encontrar e resolver possíveis problemas rapidamente. Depois que o desenvolvedor conclui uma tarefa que pode compartilhar o código com a equipe, seguindo um processo disciplinado para garantir suas mudanças, não adicionando problemas, podendo integrar com o resto do código.
- *Continuous deployment* (Implementação contínua) para SATO (2014) é uma prática de colocar cada *commit* em produção, o que geralmente significa várias implantações em produção todos os dias.

As imagens a seguir representam parte do ciclo de vida da integração (Figura 2) e implementação (Figura 3) contínua do DevOps:

Figura 2: Continuous Integration

Fonte: Adaptado de MATSUMOTA, 2018

Figura 3: Continuous Deployment

Fonte: Adaptado de MATSUMOTA, 2018

SACOLICK (2018) complementa dizendo que as práticas de DevOps também incluem:

- Controle de versão e estratégias de ramificação.
- *Containers* que padronizam e isolam os ambientes de tempo de execução do aplicativo.
- *Infrastructure as Code* (IAC), que permite criar scripts na camada de infraestrutura.
- Monitorando os pipelines Devops e a integridade dos aplicativos em execução.

Controle de versão, de acordo com Shapiro (2020) são uma categoria de ferramentas de software que ajudam equipes de software a gerenciar alterações no código-fonte com o passar do tempo. *Containers*, também conhecidos como containers linux, de acordo com PRADO (2019) é uma instância da camada de usuário do Linux, que aloca muitos recursos (CPU, memória, E/S) e é executada de forma isolada do resto do sistema. Um contêiner pode executar apenas um aplicativo ou todo o sistema de arquivos (rootfs), para que vários contêineres possam ser iniciados e executados ao mesmo tempo. IAC é um método de automação de infraestrutura de TI, utilizado principalmente para permitir que equipes de DevOps gerenciem e configurem a infraestrutura de forma automatizada por meio de código, sem a necessidade de recorrer a acesso físico ao hardware, ou mesmo através de um portal ou ferramenta de configuração (AMARAL, 2018). SACOLICK (2018) ressalta que à medida que as organizações investem em automação, uso de contêineres, padronização e implantação de aplicativos, é uma prática recomendada fazer investimentos paralelos em monitoramento.

Existem algumas ferramentas que auxiliam em tais práticas, como exemplifica a figura a seguir (Figura 4).

Figura 4: Ferramentas para práticas DevOps

Fonte: Adaptado de MATSUMOTA, 2018

De acordo com as necessidades específicas de projetos relacionados, a cultura DevOps combina uma variedade de atividades e métodos para alcançar os melhores resultados: IAC, metodologia ágil, testes, etc. Ao unificar o desenvolvimento e as operações, o DevOps pode ser dividido em quatro pilares principais: comunicação, colaboração, automação e monitoramento, como exemplifica a Figura 5.

Figura 5: Abrangência do DevOps

Fonte: Adaptado de BARBOSA, 2019

Como a cultura DevOps é um fenômeno em expansão, mas cuja adoção ao nível das organizações ainda está numa fase embrionária, necessita-se de mais investigação no sentido de clarificar os benefícios, custos e barreiras à adoção do mesmo.

1.4 O que é framework?

De acordo com ZUCHER (2020), *framework* é um termo da língua inglesa que se feita a tradução literal da palavra, significa estrutura. Mas para o desenvolvimento, *framework* é uma estrutura genérica de código para prover uma nova funcionalidade. O autor também afirma que *framework*, para desenvolvedores, que desejam aumentar sua produtividade, é um dos melhores recursos, pois ao utiliza-lo, é possível complementar o que já foi produzido até agora com os códigos genéricos que foram comentados anteriormente.

1.4.1 RECURSOS E FUNCIONALIDADES PRINCIPAIS

JOUKOVSKI (2021) lista alguma das funções do *framework* que são muito úteis, sendo algumas delas:

- Utilização de sistema modular para gerenciamento de dependências;
- Diferentes formas de conexão;
- Acesso a banco de dados relacionais;
- Motor próprio de templates para criação de interfaces;
- Programas e serviços para facilitar a publicações e manutenção de sistemas criados;
- Implementação nativa do Composer;
- Facilita a escalabilidade do sistema;
- documentação útil e organizada;
- Por ser popular, é mais fácil de encontrar soluções na internet;

O uso de framework possui suas vantagens e desvantagens, ROVEDA (2021) destaca algumas delas:

- Vantagens
    - Aumento da produtividade
    - Código sem erros

- Fácil manutenção
- Maior segurança
- Projeto padronizado
- Redução de custos
- Redução de tempo

- Desvantagens
  - Causa dependência
  - Dificuldade em configurar
  - Mais códigos que o necessário

1.5 O que é versionamento?

EVEO (2020) explica que versionamento é um processo adotado para rastrear o número de alterações em cada nova versão. Este método é muito útil para programadores para que estejam sempre cientes das mudanças feitas no software para fornecer o uso de novas funcionalidades. Por meio do controle de versão, cada nova versão do programa, hardware, driver, firmware, arquivo e etc. recebe um número exclusivo. Sempre que uma atualização é implementada, o número aumentará para identificar a versão mais recente, portanto, os usuários podem comparar as versões instaladas em seus computadores ou telefones celulares e atualizá-las quando necessário.

Usar versionamento possui suas vantagens, algumas são citadas por BATAGINI (2020), sendo elas:

- Controle de histórico
- Trabalho em equipe
- Marcações e resgate de versões estáveis
- Ramificação do projeto
- Segurança
- Confiança

1.5.1 COMO FUNCIONA.

Segundo NUNES (2020) os números de versão geralmente são atribuídos em ordem crescente e correspondem a atualizações, inclusões de ferramentas e recursos ou exclusões para

melhorar o seu funcionamento. Alguns softwares possuem números de versão internos (para maior controle dos desenvolvedores) que diferem dos números de versão do produto (para conhecimento do usuário).

NUNES (2020) afirma ainda que um esquema de versionamento de software bem montado deve usar identificadores baseados em sequências, em que cada versão é fornecida com um ou mais números e/ou letras em ordem crescente. Os primeiros números representam as mudanças com o maior nível de significância. Depois disso, eles vão demonstrando pequenas atualizações. Veja o exemplo abaixo:

- v1.2 para v1.3: pode indicar uma alteração importante na estrutura do software, como a inclusão e/ou exclusão de ferramentas;
- v1.01 para v1.02: pode representar uma pequena correção de bugs (falhas operacionais).

Esse esquema também pode usar o número "0" na primeira sequência para representar status alfa, "1" para status beta, "2" para candidato à liberação e "3" para lançamento no mercado.

1.5.2 O QUE É GIT?

ROVEDA (2021) explica que GIT é um sistema de controle de versões distribuído. Esses sistemas possuem como objetivo principal registrar quaisquer alterações feitas no conteúdo monitorado, armazenando essas informações e permitindo, caso necessário, retroceder as alterações de modo fácil e rápido. Utilizar um sistema como esse também facilita a distribuição do projeto com a equipe.

ONE (2019) cita as 3 (três) principais vantagens de usar o GIT, sendo elas:

- Performance
- Segurança
- Flexibilidade

O GIT também otimiza o desenvolvimento, fazendo isso, de acordo com ONE (2019), através de sua arquitetura e suas diversas funcionalidades e ferramentas, como por exemplo o *branching*.

- *Branching* ou Método *Branch* é uma ramificação isolada do projeto principal, para que você possa realizar testes ou implementar novas funcionalidades sem impactar na ramificação que está a parte principal do projeto. Caso esses testes

ou essa nova funcionalidade não apresente problemas, é possível enviar as novas mudanças para o projeto principal através do *merge*.

1.5.3 FERRAMENTAS DISTRIBUIDAS DE DESENVOLVIMENTO

O GitLab, de acordo com AUGUSTO (2020), é uma plataforma de hospedagem de código que permite desenvolvedores trabalharem em projetos públicos ou privados. RAVOOF (2021) complementa informando que o GitLab também e um plataforma Git e DevOps em nuvem, ajudando os mantenedores a monitorar, testar e implementar o código.

Uma das principais funcionalidades que se destaca no GitLab, de acordo com BERTOLA (2019), é o foco que a ferramenta vem dado à integração a ferramentas de DevOps. Ela nativamente proporciona ferramentas de integração e entrega contínua ou CI/CD, além de disponibilizar métricas para acompanhar a qualidade de código e performance por exemplo.

RAVOOF (2021) reitera em seu texto que caso esteja procurando começar no DevOps, o GitLab é a escolha mais ponderada e provavelmente mais barata, tendo em vista que pode-se começar de graça, independente de nenhum outro serviço pago, tornando-se a ferramenta ideal para o cenário tratado no presente documento.

1.6 Docker

GOMES (2018) explica que o Docker e uma ferramenta Open Source (código aberto), desenvolvida na linguagem de programação Go. O mesmo por ser de alto desempenho, facilita a criação e administração de ambientes isolados, diminuindo os riscos e aumentando a velocidade da disponibilização do programa para o usuário final. GUEDES (2018) também diz que ao utilizar a ferramenta e sua conteinerização, torna a aplicação portátil entre hosts que possuam o Docker instalado, conseguindo criar, implantar, copiar e migrar entre ambientes facilmente.

DALMAZO (2021) informa algumas vantagens que usar esse tipo de ferramenta traz, citando alguns pontos, como por exemplo:

- Economia de recursos
- Maior disponibilidade do sistema
- Compartilhamento
- Facilidade de gerenciamento

- Ambientes similares
- Padronização e replicação

1.7 Visual Studio

LEE (2021) afirma que a IDE (Integrated Development Environment) é um programa rico em recursos que oferece suporte a muitos aspectos do desenvolvimento de software. O IDE do Visual Studio é um painel de inicialização criativo que pode ser usado para editar, depurar e codificar e, em seguida, publicar o aplicativo. Além dos editores e depuradores padrão fornecidos pela maioria dos IDEs, o Visual Studio também inclui compiladores, ferramentas de conclusão de código, designers gráficos e muitos outros recursos para melhorar o processo de desenvolvimento de software.

2 METODOLOGIA

Fora realizado um estudo das práticas contidas em um processo DevOps, dedicando-se à compreensão de seus pontos de contribuição na comunicação, colaboração, automação e monitoramento em prol da melhoria da qualidade no desenvolvimento, mas também suas dificuldades de aplicação e estruturação para serem colocadas em vigor. Em paralelo foram realizadas pesquisas bibliográficas que tratam da criação e estruturação de processos ágeis, tendo como objetivo consolidar uma base de conhecimento mais amplo sobre os processos de desenvolvimento de software.

Assim, a segunda etapa do projeto consistiu na análise dos processos comuns de desenvolvimento e operações dentro das empresas, assim como a interação entre as respectivas equipes de desenvolvimento e operações, para se configurar o ambiente de desenvolvimento com as principais funcionalidades. Em seguida, feita a configuração do repositório no qual armazena a configuração da pipeline, e a criação do ambiente de DevOps.

Dessa forma, para se medir as vantagens e avanços, se tornou necessário observar com um olhar crítico às possíveis causas e efeitos já identificados, fazendo uma comparação dos processos estudados. Tais informações foram levantadas através de testes práticos e questionários para consolidação de estatísticas ali encontradas.

A implementação do processo DevOps foi realizada visando criar um modelo mínimo de utilização das práticas estudadas utilizando como base o modelo proposto por Dias (2018), gerente de cloud da Mandic Cloud Solutions, as etapas de implementação de DevOps serão as seguintes:

Primeiro é preciso realizar mudanças na cultura dentro do ambiente de trabalho para que haja uma equipe integrada, uma vez que o DevOps visa uma maior integração entre os membros da empresa e das equipes. Trabalhar este ponto na cultura organizacional faz com que a interação entre as equipes não seja uma tarefa a ser feita e sim uma rotina de trabalho.

Em seguida a análise dos impactos da mudança e se as metodologias ágeis utilizadas contêm a mesma visão para planejar atividades focadas em resultados, automação e integração.

Nas automatizações, a parte técnica suporta a gestão de pessoas e vice-versa. Nessa parte fora utilizada a ferramenta GitLab para integrar ferramentas e processos, criando um ambiente onde os artefatos gerados pelas equipes possam se conectar de uma maneira mais simples e fluida. Antes de realizar os testes automatizados, é indispensável a execução de testes de

regressão, aceitação, integração, entre outros, para garantir que o ciclo automatizado criado está adequado.

Utilizar métricas e objetivos para especificar e planejar o que será utilizado durante o projeto e realizar o mapeamento da evolução dos projetos faz-se fundamental para a evolução da proposta, uma vez que a resposta do ambiente é a base para saber em quais pontos houveram melhorias e onde é preciso reajustar.

Passando pelas etapas, serão observadas as respostas das equipes quanto a produtividade e a qualidade dentro das sprints criadas, validando assim cada uma das etapas de implantação da cultura DevOps, podendo haver novas melhorias e adaptações caso seja observada necessidade.

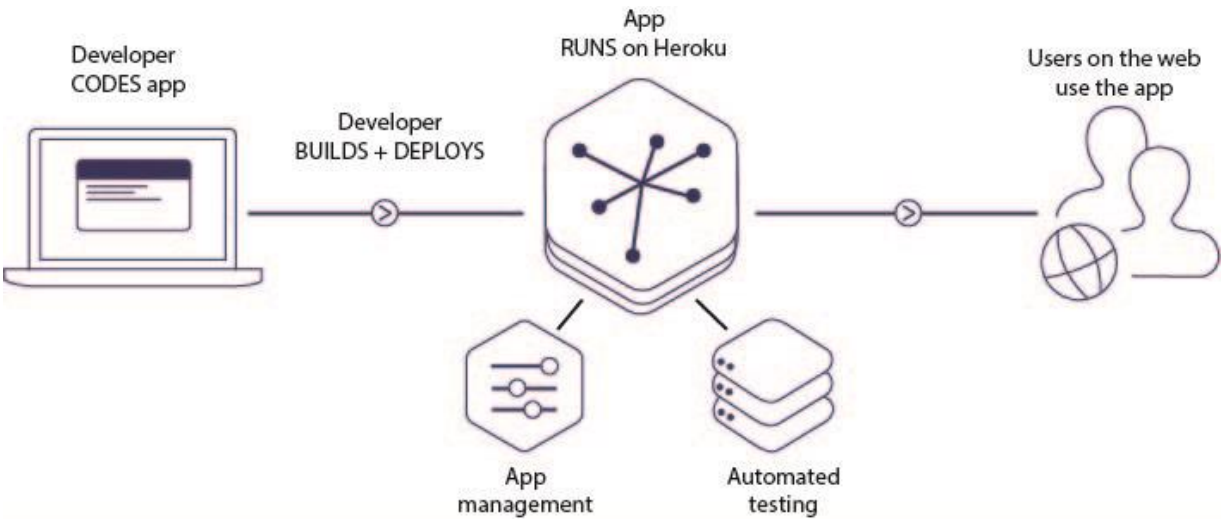
3 RESULTADOS

3.1 Processo DevOps

Após um estudo sobre as principais práticas de DevOps, foi possível observar uma estrutura mínima para a criação de um ambiente. A ideia do processo é automatizar tarefas simples, mas que antes estavam fortemente interligadas a ação humana, tanto o processo de teste quanto o deploy, sendo assim o ponto principal é não haver mais uma dependência direta do conhecimento do time e permitir que o processo de conhecimento seja contínuo.

Com base nos objetivos de simplificação de um processo demorado e repetitivo, a automação contempla a possibilidade de um desenvolvimento contínuo para uma constante evolução e ainda assim mantendo a transparência de código com os integrantes de equipe, para uma comunicação clara e facilitando o reconhecimento de um erro futuro, abaixo a imagem do processo completo (Figura 6).

Figura 6: Processo mínimo DevOps



Fonte: Autores

A estrutura do processo é simples, quando o desenvolvedor termina de estruturar o código da funcionalidade na qual está trabalhando ele submente a todo o processo de build e deploy, processo esse que será executado na estrutura do ambiente criado, gerenciando a aplicação e executando testes automatizados na funcionalidade, em seguida a funcionalidade é enviada para o sistema do cliente, com todo o processo sendo feito de forma automatizada, explicaremos em detalhes a seguir.

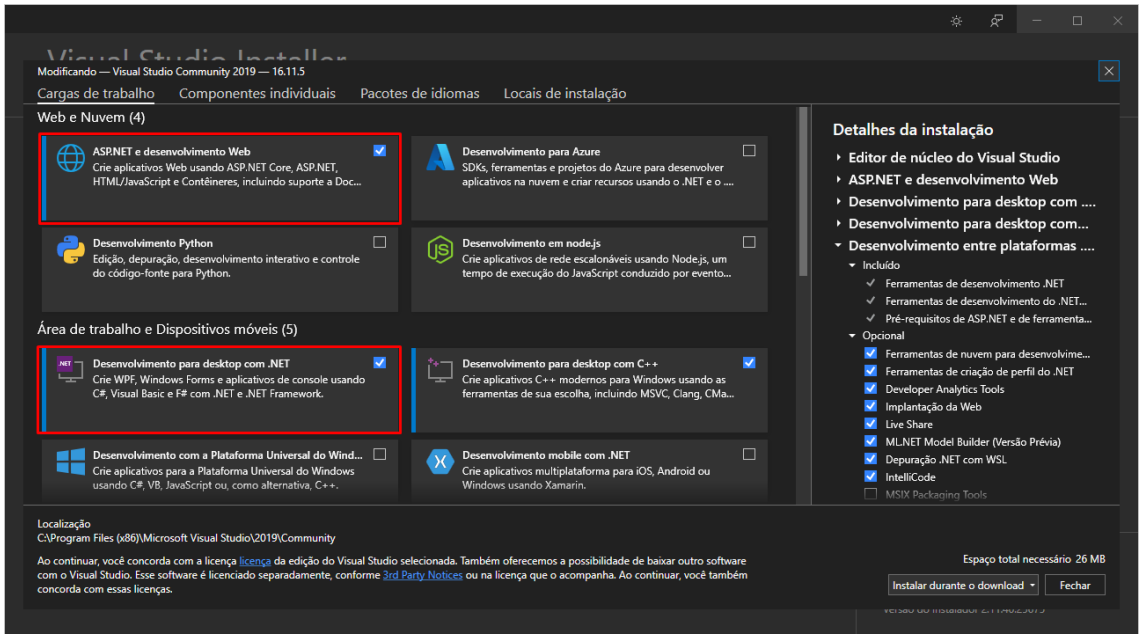
3.2 Estruturação do ambiente

Para que houvesse uma melhor aplicação dos testes foi feito um estudo sobre uma necessidade real de desenvolvimento para criar um código que sirva como base para testes simulando um ambiente de desenvolvimento comum nas empresas, para isso foi levantado as principais funcionalidades de um sistema, criando assim um MVP do projeto, mantendo a complexidade, mas permitindo que fosse viável construí-lo no tempo proposto.

Para configurar o ambiente de desenvolvimento, inicialmente faz-se necessário instalar o Git. A forma de instalação depende do Sistema Operacional, para demonstração neste projeto foi utilizando o sistema operacional Windows.

A segunda ferramenta que deve ser preparada é *Integrated Development Environment* (IDE), a IDE utilizada pode ser a de preferência do desenvolvedor, neste trabalho será demonstrado através do Visual Studio por ser uma ferramenta poderosa e com várias funcionalidades que ajudam o trabalho em C# e .NET Core (linguagem utilizada para estudo) além de ser uma ferramenta que os autores possuem domínio. Para isso, recomendo instalar os complementos destacados (Figura 7 e 8):

Figura 7: Complementos de instalação (Parte 1)



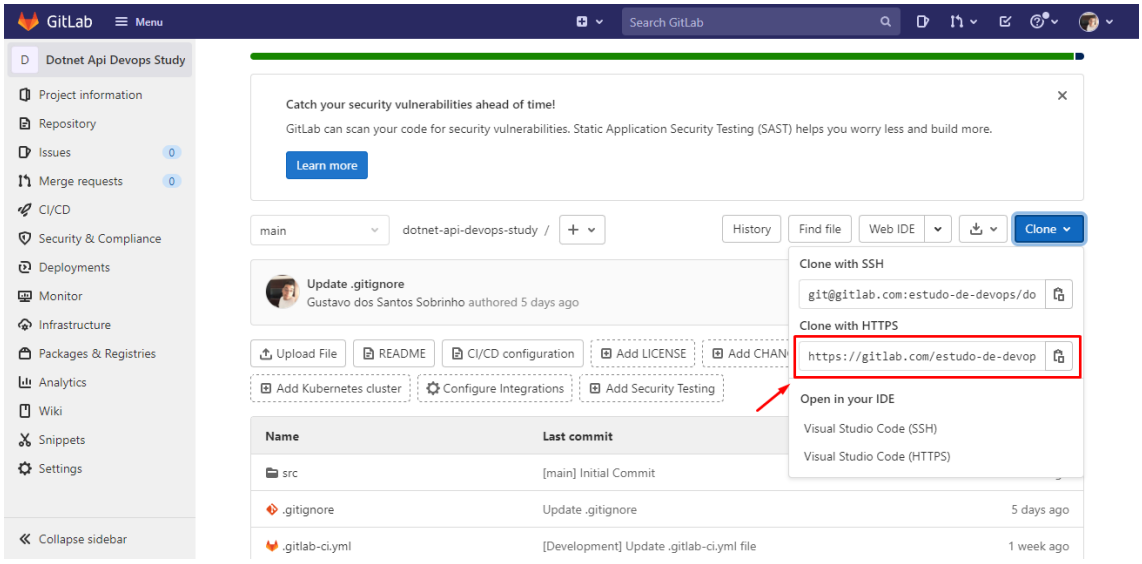
Fonte: Autores

Figura 8: Complementos de instalação (Parte 2)

Fonte: Autores

Após esses passos, é preciso clonar o repositório, ou seja, fazer uma cópia do repositório que será configurado no próximo tópico, para isso e só utilizar o seguinte comando: git clone [linkDoRepositório], onde link do repositório pode ser consultado no repositório GitLab (Figura 9):

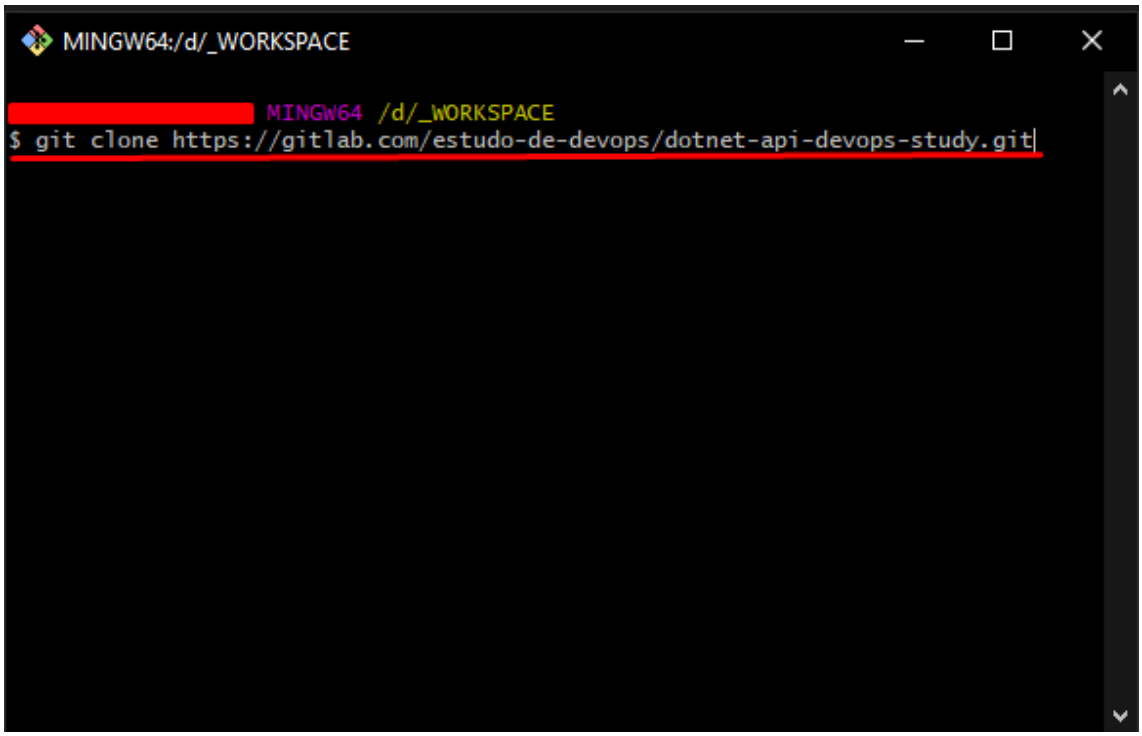
Figura 9: Repositório



Fonte: Autores

Para utilizar o comando citado, deve-se utilizar o prompt de comandos do Git, o GitBash (Figura 10).

Figura 10: Prompt Git Bash

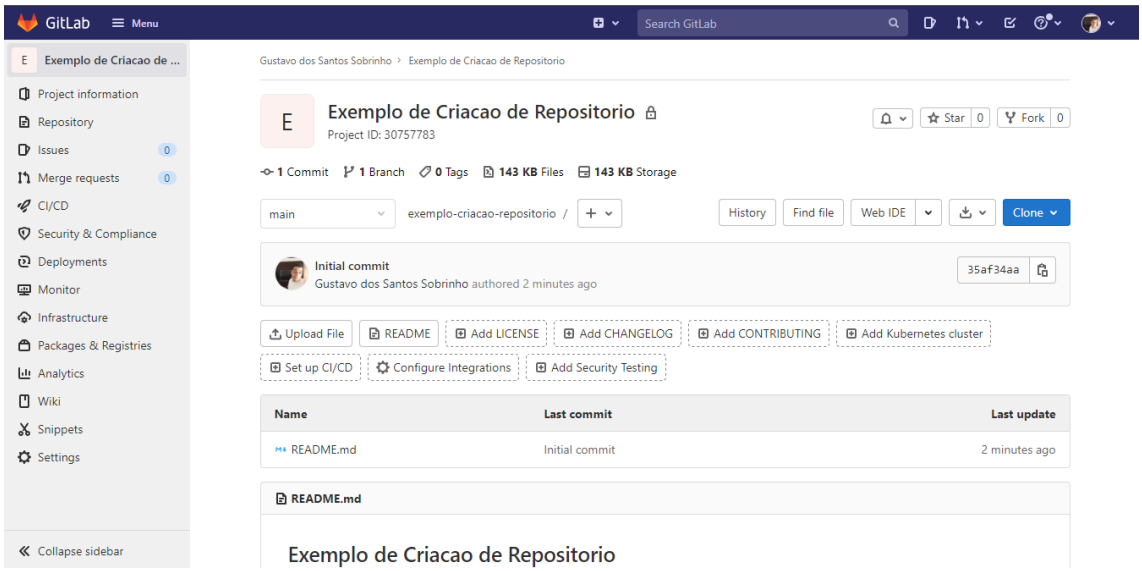


Fonte: Autores

Parar criar um repositório no GitLab é bem intuitivo, basta clicar em *New Project* (Novo Projeto), escolher dentre uma das quatro opções, *Create blank project* (Criar Projeto Vazio), *Create from template* (Criar a partir de um modelo), *Import Project* (Importar projeto), *Run CI/CD for external repository* (Executar CI/CD em um repositório fora do GitLab).

Para exemplo foi criado um projeto vazio, deixando marcada as opções padrões do repositório, tendo como único campo alterado foi o nome do repositório. A partir desse momento seu repositório já está criado, ele estará parecido com a seguinte imagem (Figura 11).

Figura 11: Repositório novo

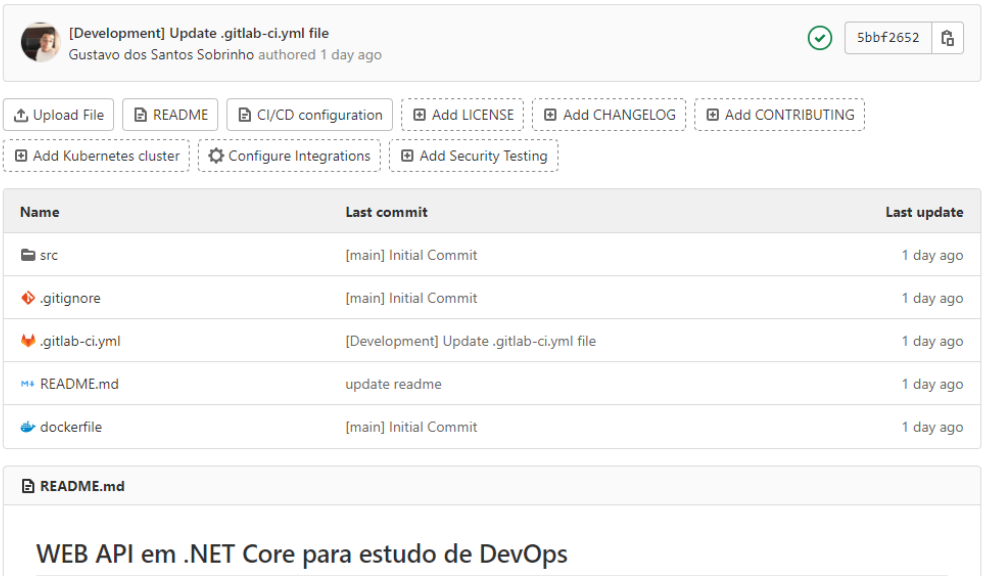


Fonte: Autores

Na criação do projeto, deixamos a opção de projeto privado marcado, isso significa que apenas as pessoas que concebermos permissões poderão ter acesso ao repositório. Para adicionar novos membros, basta clicar em *Project Information* e depois clicar em *members*, ambos no menu lateral da esquerda. Ao "convidar" novos membros para colaborar com o repositório, você consegue definir a função, onde pode escolher entre *Guest*, *Reporter*, *Developer* e *Mantainer*, cada função é explicada com detalhes na própria documentação disponibilizada pela ferramenta, disponível em https://docs.gitlab.com/ee/user/permissions.html.

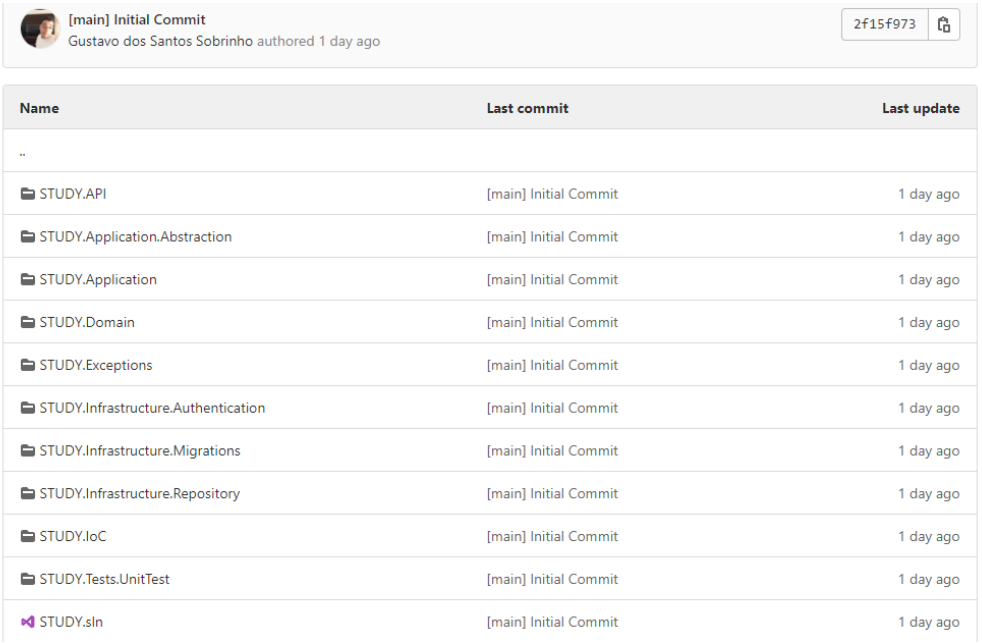
Nesse momento, será detalhado a criação do DockerFile e configuração das *pipelines*. Para as configurações a seguir, se depende da estrutura de pastas, será explicado com todos os arquivos necessários para o funcionamento do código dentro da pasta "src", como mostra as imagens a seguir (Figura 12 e 13).

Figura 12: Distribuição de pastas (Parte 1)



Fonte: Autores

Figura 13: Distribuição de pastas (Parte 2)



Fonte: Autores

Os arquivos de configurações estão fora da pasta “src” como mostra a Figura 12. O arquivo .gitignore contém arquivos e pastas que serão ignoradas na hora do *commit* e do *push*, é possível encontrar diversos exemplos de sua criação para cada linguagem de programação. O

DockerFile é onde será feita a configuração da imagem do seu container. Vamos explicar o que o arquivo está fazendo baseado no código a seguir:

```
1  FROM mcr.microsoft.com/dotnet/sdk:3.1 AS build
2
3  WORKDIR ./
4
5  COPY ./src ./src
6
7  WORKDIR "./src/STUDY.API"

8  RUN dotnet build -c Release -o /app/build

9  FROM build AS publish
10 RUN dotnet publish -c Release -o /app/publish
11
12 FROM mcr.microsoft.com/dotnet/aspnet:3.1
13 ENV ASPNETCORE_URLS=http://+:8080
14
15 WORKDIR /app
16 COPY --from=publish /app/publish .
17 CMD ASPNETCORE_URLS=http://*:$PORT dotnet STUDY.API.dll
```

Fonte: Autores

Na linha 1, é definida a linguagem que será utilizada para a imagem do container interpretar o código, nesse caso .NET 3.1. Na linha 3, você define a pasta que a imagem interpretara os comandos dali pra frente, como o DockerFile está em uma pasta fora do “src” definimos que ele começará a trabalhar a partir de onde ele está, logo em seguida, na linha 5, como o nosso código fonte está dentro da pasta “src” (imagem 8), pedimos pra ele copiar tudo que está em “src” para uma pasta da imagem também chamada “src”. Na linha 7 mudamos a pasta de trabalho para a pasta onde está nosso projeto principal, que será explicado com mais detalhes no próximo tópico. Na linha 8, executamos um comando para compilar o projeto, com a saída do resultado na pasta “/app/build”. Na linha 9 damos o “apelido” de publish para o comando “build” para melhor compreensão e a saída do resultado vai para a pasta “/app/publish”. Na linha 13 definimos as URLS que serão disponibilizadas para acesso do Docker. Na linha 16 ele copia o que está na pasta “/app/publish”. Na linha 17 ele define uma rota e executa o projeto.

Agora vamos explicar o .gitlab-ci.yml com base no código abaixo:

```
image: mcr.microsoft.com/dotnet/core/sdk:3.1

stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - dotnet build ./src/

test_Job:
  stage: test
  script:
    - 'dotnet test ./src/ --
logger:"junit;LogFilePath=..\artifacts\{assembly}-test-
result.xml;MethodFormat=Class;FailureBodyFormat=Verbose"'
  artifacts:
    when: always
    paths:
      - ./**/*test-result.xml
    reports:
      junit:
        - ./**/*test-result.xml

deploy:heroku:
  stage: deploy
  image: docker
  services:
    - docker:dind
  before_script:
    - apk update
    - apk upgrade
    - apk add nodejs
    - apk add bash
    - apk add curl
    - node --version
    - curl https://cli-assets.heroku.com/install.sh | sh

  script:
    - echo "> pushing to heroku"
    - echo $NETRC > ~/.netrc
    - echo -n $CI_REGISTRY_TOKEN | docker login -u gitlab-ci-token --
password-stdin $CI_REGISTRY
    - docker build -f dockerfile -t $CI_REGISTRY/$CI_REGISTRY_APP/web
.
    - docker push $CI_REGISTRY/$CI_REGISTRY_APP/web
    - heroku container:release web --app=$CI_REGISTRY_APP

  dependencies:
    - test_Job

  only:
    - Development
```

Fonte: Autores

Quando é inserido o código “image: mcr.microsoft.com/dotnet/core/sdk:3.1” você está definindo a linguagem que a pipeline vai interpretar os comandos. Após isso, você define as etapas que sua pipeline tem q passar até o *deploy* do projeto, Figura 14.

Figura 14: Etapas da pipeline

```
stages:
  - build
  - test
  - deploy
```

Fonte: Autores

Após definir as etapas, você define o nome que aparecerá no pipeline, por exemplo, “*build_job*” e o que ela fará. “Dentro” do *build_job* (Figura 15), você passa qual etapa ele vai executar, nesse caso, a etapa de build. A palavra reservada “*script*” irá executar o comando de compilar o projeto que está na pasta “src” para checar se possui algum erro, caso possua, a pipeline não prossegue.

Figura 15: Etapa de build

```
build_job:
  stage: build
  script:
    - dotnet build ./src/
```

Fonte: Autores

Logo em seguida é a etapa de testes, o nome dado foi “*test_Job*” (Figura 16), segue o mesmo princípio do “*build_job*”, porém, ele vai gerar artefatos, e esses artefatos serão coletados para saber se todos os testes foram executados com sucesso, caso tenha gerado algum erro, a partir desse momento a pipeline não executa mais. Esses artefatos serão gerados de acordo com a frequência definida para o argumento “*when*”.

Figura 16: Etapa de teste

```
test_Job:
  stage: test
  script:
    - 'dotnet test ./src/ --logger:"junit;LogFilePath=..\artifacts\{assembly}-test-result.xml;MethodFormat=Class;FailureBodyFormat=Verbose"'
  artifacts:
    when: always
    paths:
      - ./**/*test-result.xml
    reports:
      junit:
        - ./**/*test-result.xml
```

Fonte: Autores

Agora, por último, é executado o *deploy* no servidor escolhido, nesse caso, escolhemos o Heroku pois é uma ferramenta gratuita que atende muito bem a necessidade. Para essa etapa demos o nome "*deploy:heroku*" (Figura 17).

Figura 17: Etapa de *deploy*

```
deploy:heroku:
  stage: deploy
  image: docker
  services:
    - docker:dind
  before_script:
    - apk update
    - apk upgrade
    - apk add nodejs
    - apk add bash
    - apk add curl
    - node --version
    - curl https://cli-assets.heroku.com/install.sh | sh

  script:
    - echo "> pushing to heroku"
    - echo $NETRC > ~/.netrc
    - echo -n $CI_REGISTRY_TOKEN | docker login -u gitlab-ci-token --password-stdin $CI_REGISTRY
    - docker build -f dockerfile -t $CI_REGISTRY/$CI_REGISTRY_APP/web .
    - docker push $CI_REGISTRY/$CI_REGISTRY_APP/web
    - heroku container:release web --app=$CI_REGISTRY_APP

  dependencies:
    - test_Job

  only:
    - Development
```

Fonte: Autores

O princípio dessa etapa segue a mesma ideia das etapas anteriores, porém, temos uma variável de "*services*", na qual passamos "docker:dind". Quando falamos "dind", nos referimos a *Docker-in-Docker*, ou seja, um container Docker com uma instalação Docker que permite executar outros containers dentro dele. Antes de executar a publicação em si, vamos atualizar os pacotes do container que será responsável por executar a compilação do meu container Docker, por isso, existe um processo "*before_script*", que executará essa atualização antes de executar os scripts abaixo, isso é possível por causa do docker:dind. Após a atualização, ele executa os scripts de *deploys* (Figura 18).

A variável de "*dependencies*" registra uma dependência de uma etapa para poder executar a etapa de *deploy*, por exemplo, está configurada para depender do "*test_Job*" para executar. A variável "*only*" restringe aquela etapa para apenas uma *branch*, nesse caso, a *branch* "*Development*".

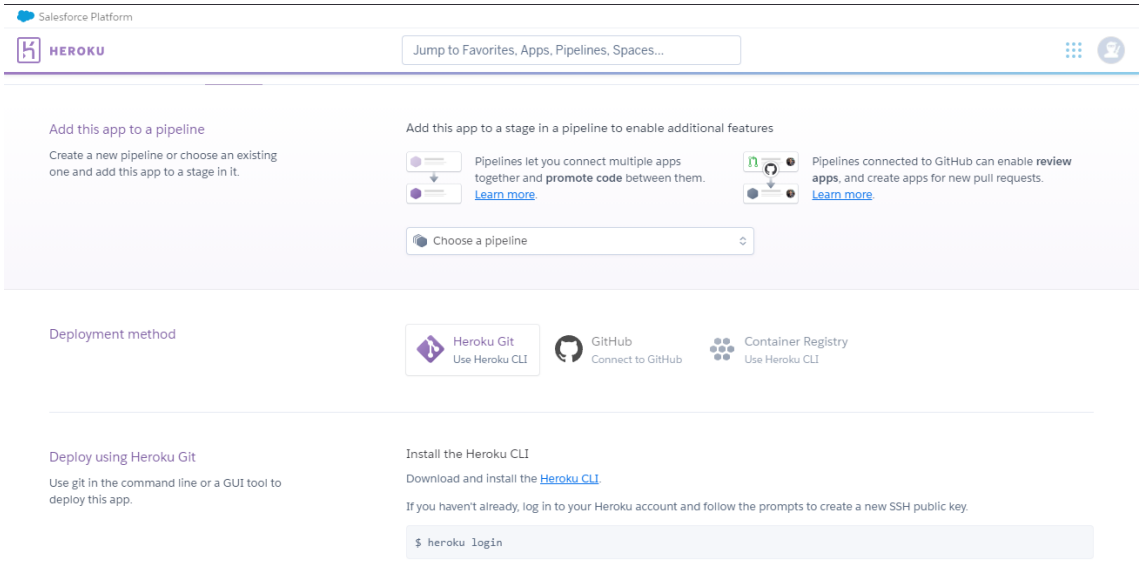
Figura 18: Scripts da etapa de deploy

```
script:
  - echo "> pushing to heroku"
  - echo $NETRC > ~/.netrc
  - echo -n $CI_REGISTRY_TOKEN | docker login -u gitlab-ci-token --password-stdin $CI_REGISTRY
  - docker build -f dockerfile -t $CI_REGISTRY/$CI_REGISTRY_APP/web .
  - docker push $CI_REGISTRY/$CI_REGISTRY_APP/web
  - heroku container:release web --app=$CI_REGISTRY_APP
```

Fonte: Autores

O termo $NETRC como as outras palavras com $ são variáveis da *pipeline* que são configuradas no GitLab. Para conseguir os valores para essas variáveis, é necessário criar uma conta e um projeto no Heroku (https://www.heroku.com/home), no qual será nosso ambiente de DevOps. Para criar um projeto no Heroku, é simples, basta clicar em *New* (Novo), depois *Create new app* (Criar novo *app*), dar um nome para o *App*, e clicar em *Create app* (Criar app). Após a criação, sua tela se parecerá com a Figura 19.

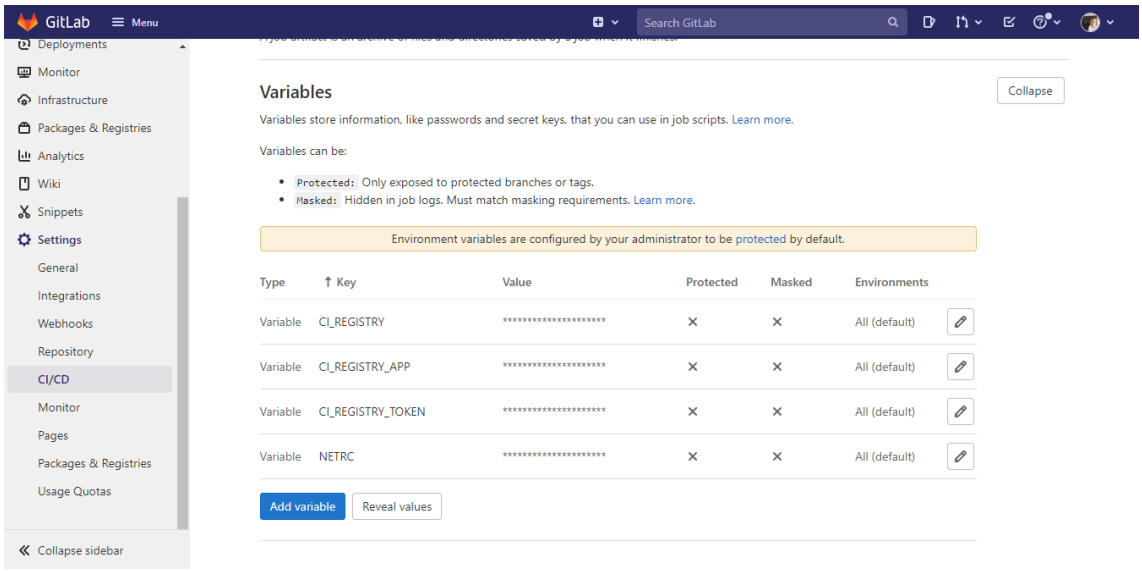
Figura 19: Aplicativo novo heroku



Fonte: Autores

Para configurar as variáveis da *pipeline* do GitLab, basta clicar em *Settings* e depois em CI/CD e procurar o tópico *Variables* (Figura 20). É importante desmarcar a *checkbox* de "*Protect variable*" para a pipeline conseguir reescrever os valores.

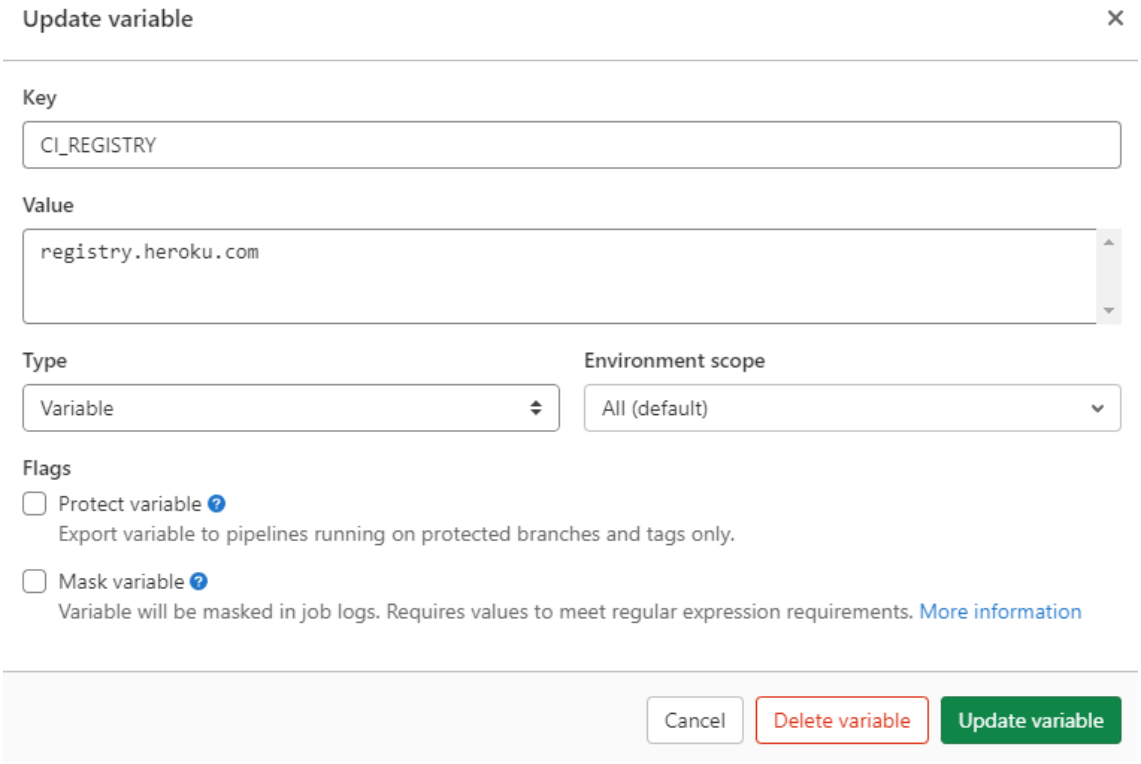
Figura 20: Variáveis da pipeline



Fonte: Autores

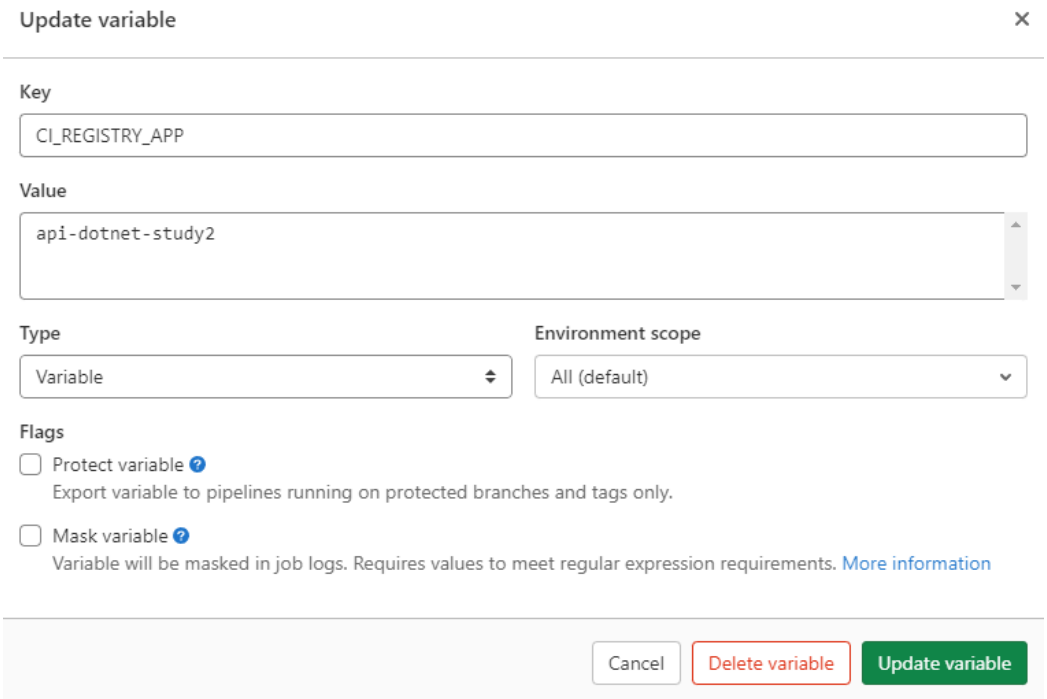
- O valor de CI_REGISTRY por padrão é registry.heroku.com (Figura 20).
- O valor de CI_REGISTRY_APP é o nome do projeto que você criou no Heroku, por exemplo api-dotnet-study2 (Figura 21).
- O valor de CI_REGISTRY_TOKEN é a API Key que o Heroku disponibiliza ao acessar as configurações da sua conta no Heroku (Figura 22).
- O valor de NETRC é a combinação do e-mail de cadastro do Heroku e a API Key (Figura 24).
    - machine api.heroku.com
      login seu-email@email.com
      password api-key
      machine git.heroku.com
      login seu-email@email.com
      password api-key

Figura 21: Variável CI_REGISTRY



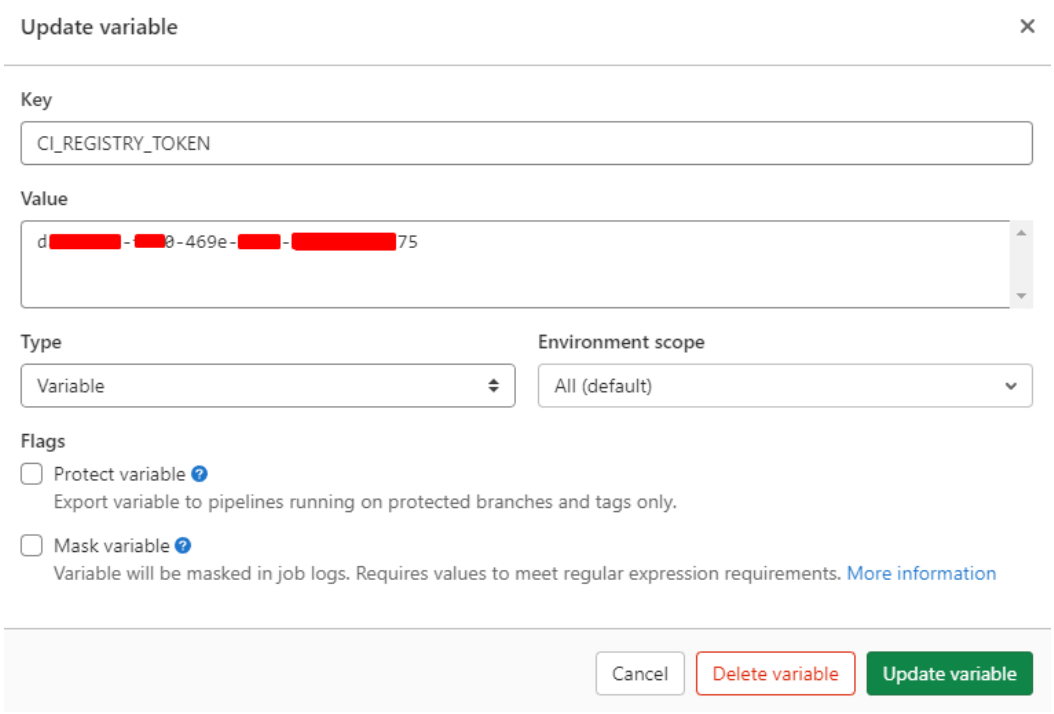
Fonte: Autores

Figura 22: Variável CI_REGISTRY_APP



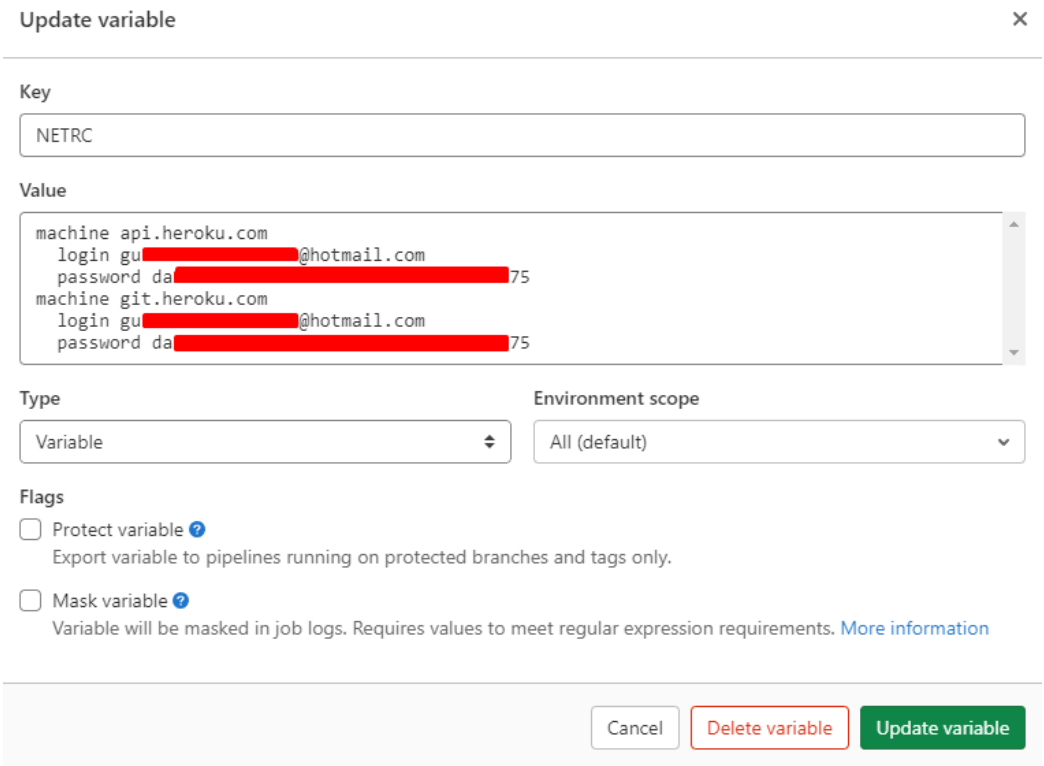
Fonte: Autores

Figura 23: Variável CI_REGISTRY_TOKEN



Fonte: Autores

Figura 24: Vaiável NETRC



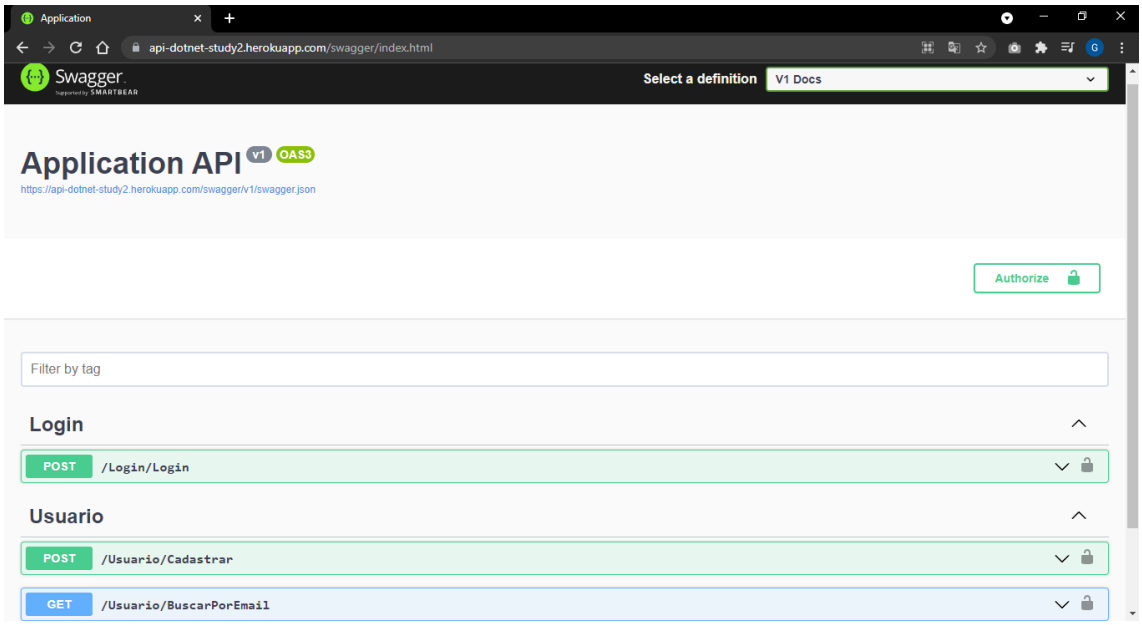
Fonte: Autores

Nesse ponto, tudo que era necessário para seu ambiente de DevOps funcionar já está pronto. Agora qualquer *commit* para a *branch* “Development” vai iniciar um *deploy* da sua aplicação no Heroku. Você pode acompanhar isso através da aba CI/CD do GitLab.

O código de exemplo é uma Web Api feito em .NET Core, um framework para linguagem de programação C#. A princípio, seu objetivo é realizar autenticação através de um e-mail e retornar um JWT Token para ser utilizado como chave de autenticação, mas para exemplo do funcionamento do DevOps, não será dado muitos detalhes.

Ao acessar o endereço onde a API está disponibilizada (https://api-dotnet-study2.herokuapp.com/swagger/index.html), percebemos 3 *endpoints* que podemos acessar (Figura 25).

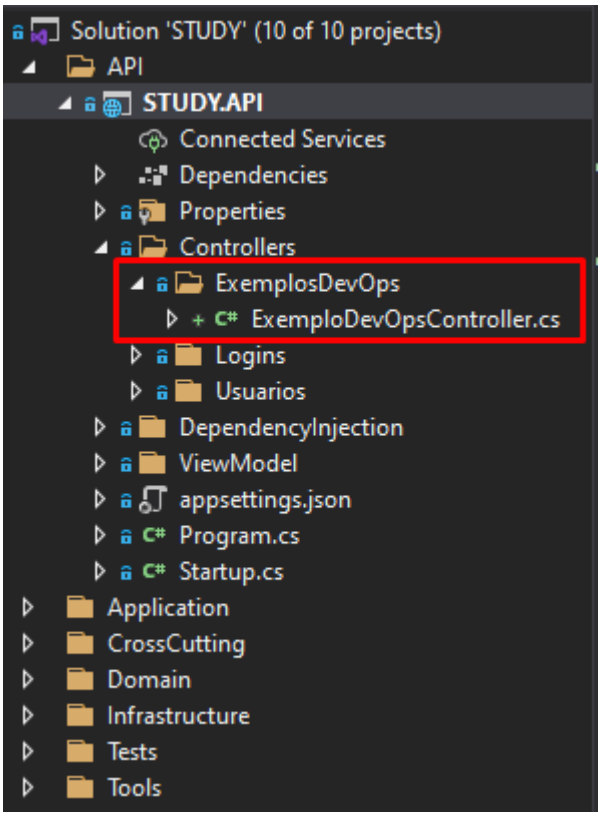
Figura 25: Endpoints no servidor pré-publicação



Fonte: Autores

Para demonstrar o funcionamento, vamos criar um outro *Controller* que não realizará nada complexo (Figura 26).

Figura 26: Controller novo



Fonte: Autores

O código contido nele apenas retornará uma mensagem com a data e a hora da execução do *endpoint* (Figura 27).

Figura 27: Endpoint criado para demontras o DevOps

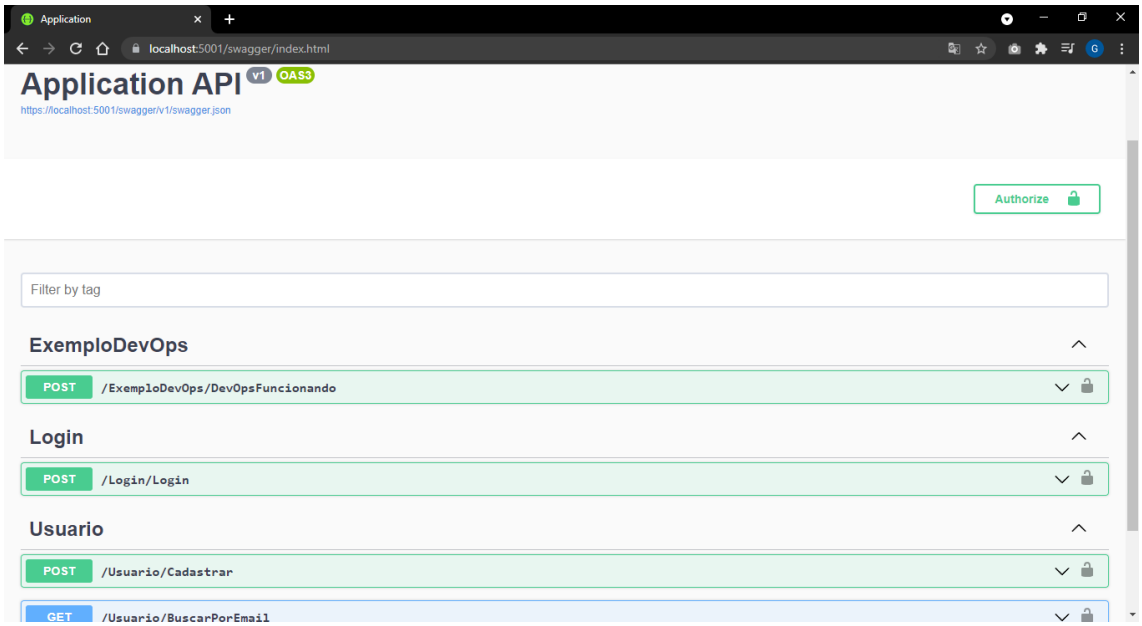
```
using Microsoft.AspNetCore.Mvc;
using System;
using System.Threading.Tasks;

namespace STUDY.API.Controllers.ExemplosDevOps
{
    [ApiController]
    [Route("[controller]")]
    0 references
    public class ExemploDevOpsController : ControllerBase
    {
        [HttpPost("DevOpsFuncionando")]
        0 references
        public async Task<IActionResult> DevOpsFuncionando()
        {
            return Ok($"O DevOps está funcionando! Teste realizado dia {DateTime.Now.ToString("dd/MM/yyyy")} as {DateTime.Now.ToString("HH:mm")}");
        }
    }
}
```

Fonte: Autores

E ao testar localmente, percebemos que ele já aparece na listagem de *endpoints* (Figura 28).

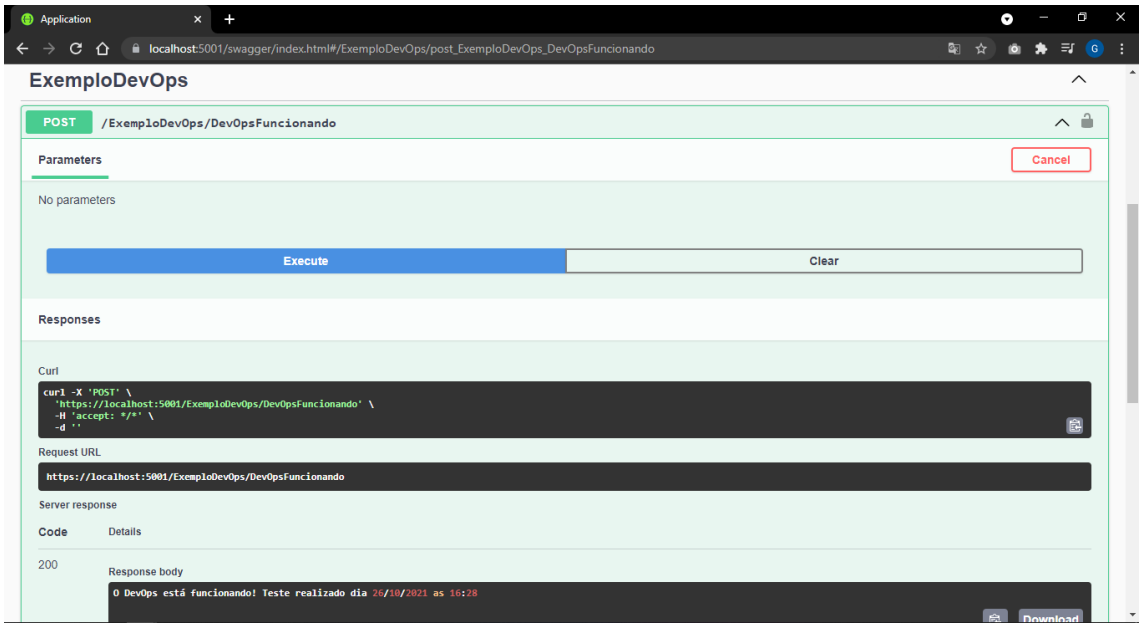
Figura 28: Endpoints no teste local



Fonte: Autores

E ao testar, ele já está retornando a mensagem que definimos no código (Figura 29).

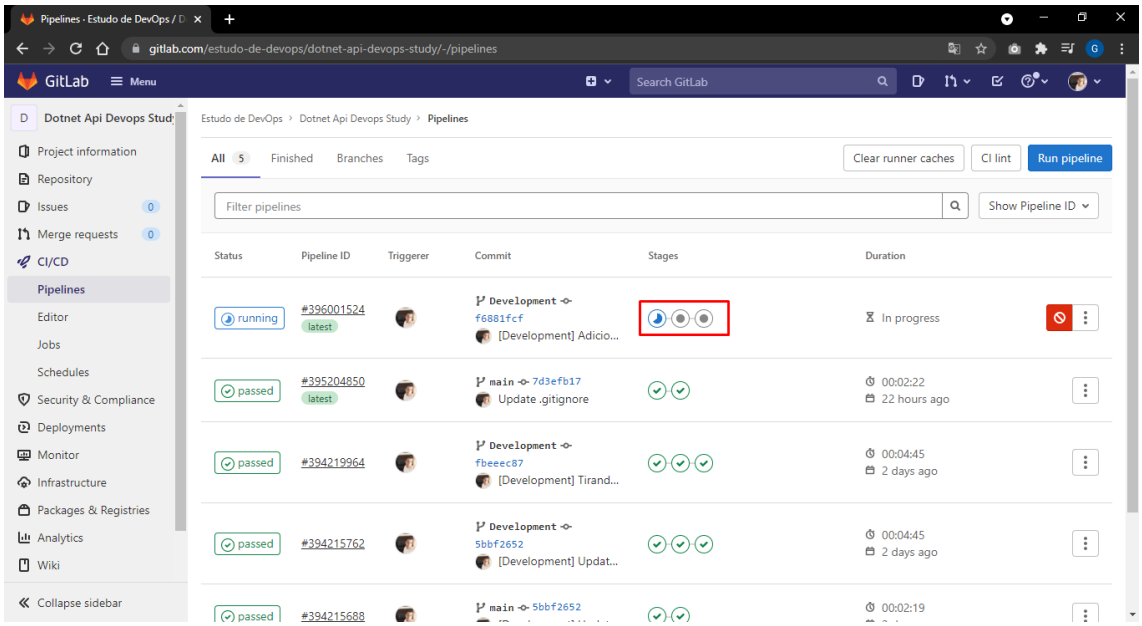
Figura 29: Resultado ao executar endpoint novo



Fonte: Autores

Como as mudanças foram feitar na *branch* “Development” (a mesma configurada no arquivo “.gitlab-ci.yml” para executar o *deploy* no Heroku), após realizar o *commit*, a *pipeline* já será executada as 3 etapas, sendo elas: *build*, *test* e *deploy* (Figura 30).

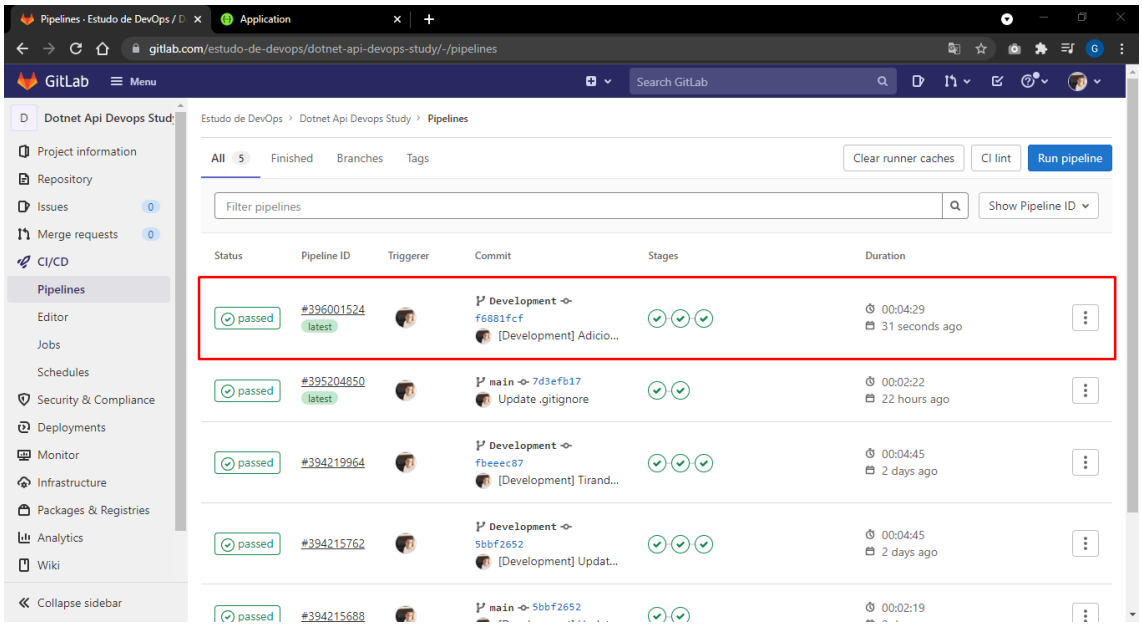
Figura 30: Etapas da pipeline



Fonte: Autores

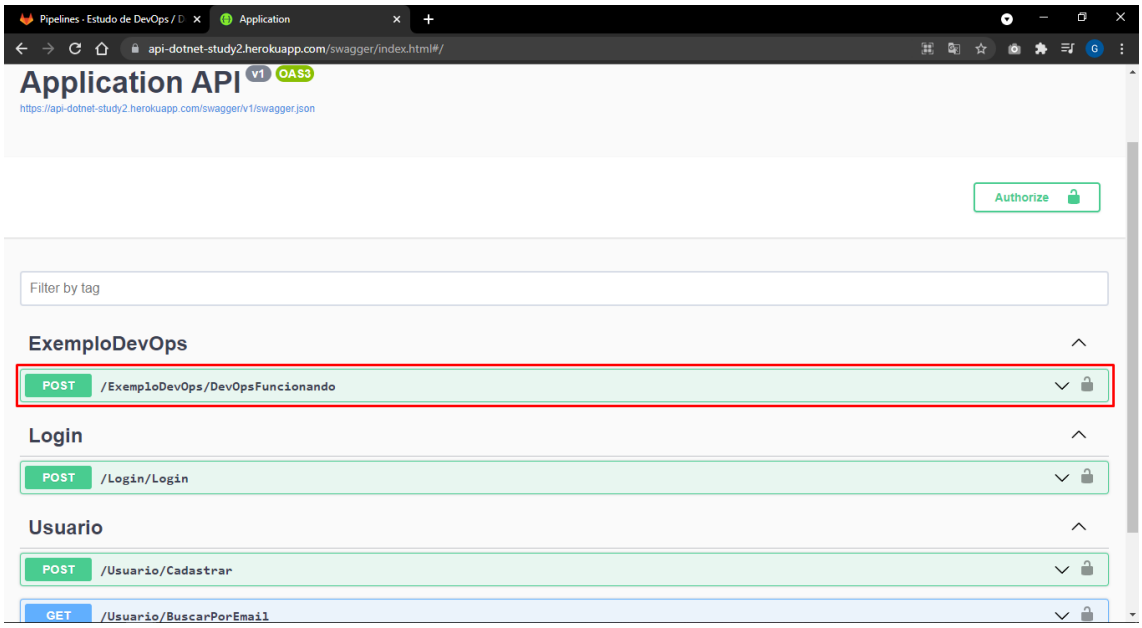
Após as 3 (três) etapas concluírem com sucesso (Figura 31) e atualizarmos o link do Heroku, o novo *endpoint* já está disponível para teste (Figura 32).

Figura 31: Etapas da pipeline completa



Fonte: Autores

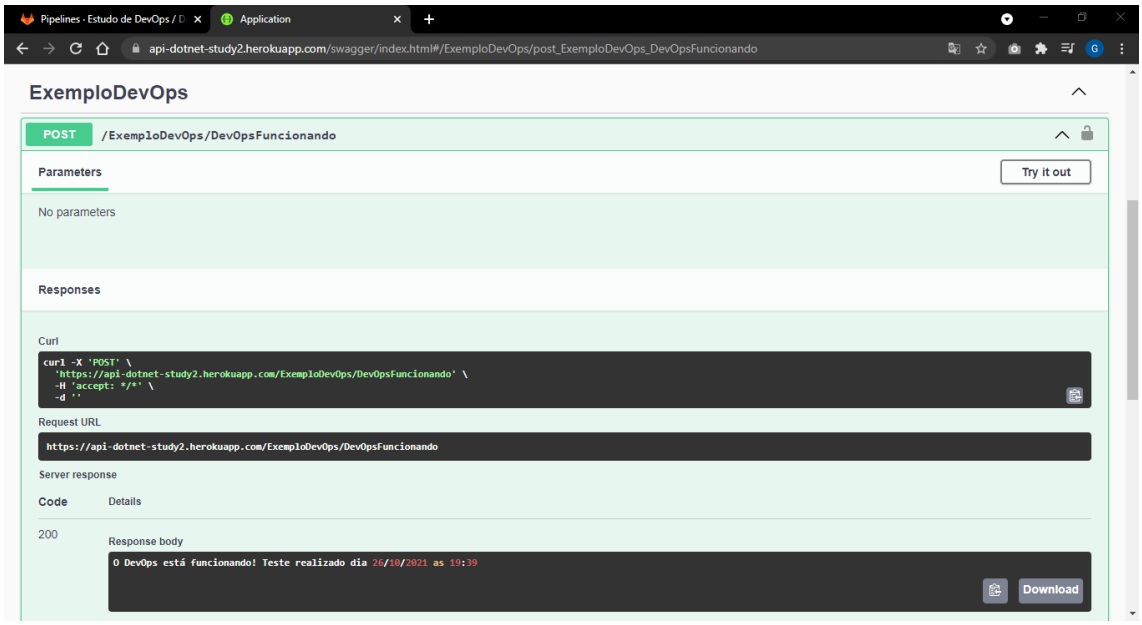
Figura 32: Endpoints no servidor pós publicação



Fonte: Autores

Ao executar, é retornada a mesma mensagem quando testada localmente, porém, com diferença de horas pois o servidor não está localizado no Brasil (Figura 33).

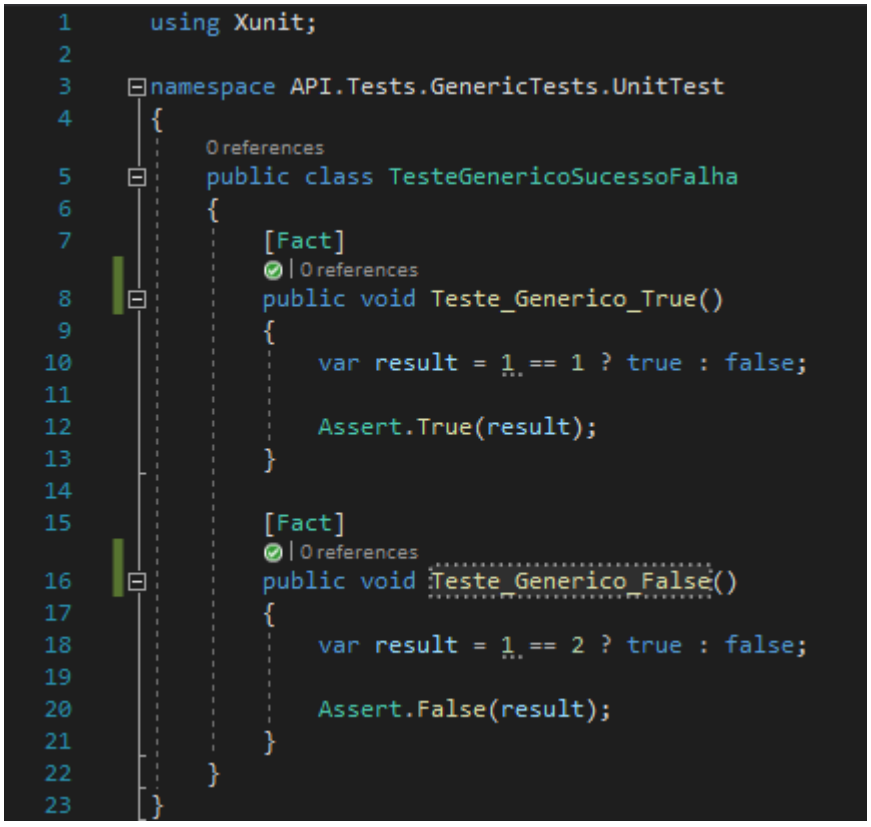
Figura 33: Resultado ao executar endpoint novo pós publicação



Fonte: Autores

Agora vamos “estragar” um teste unitário para ver o que acontece com a pipeline. Os testes atuais são apenas testes simples, apenas para título de estudo (Figura 34).

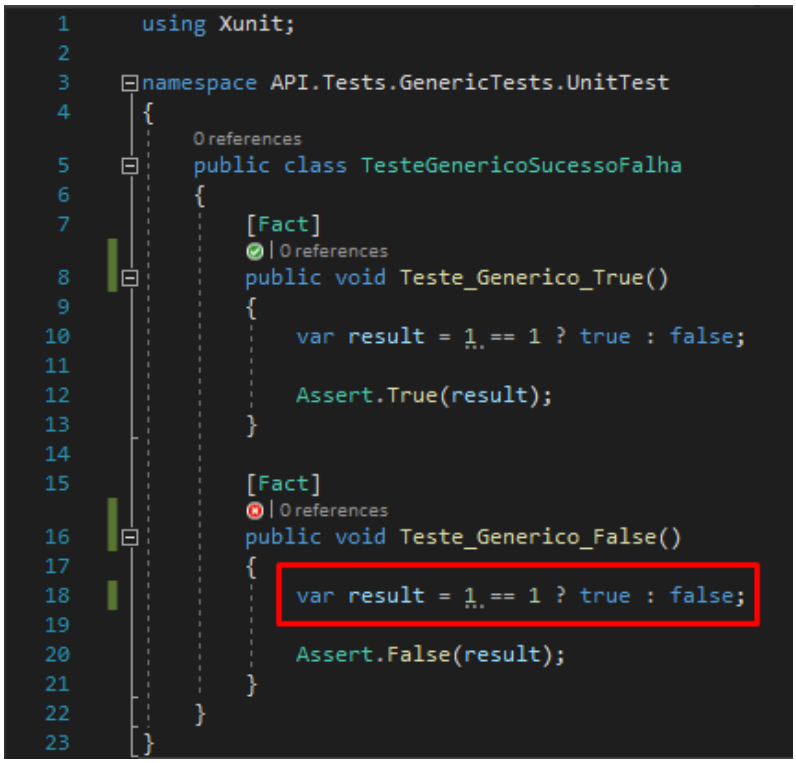
Figura 34: Testes unitários



Fonte: Autores

A linha 20, espera que o resultado seja falso, dessa forma vamos alterar o Teste_Generico_False() para o resultado retornar verdadeiro (Figura 35).

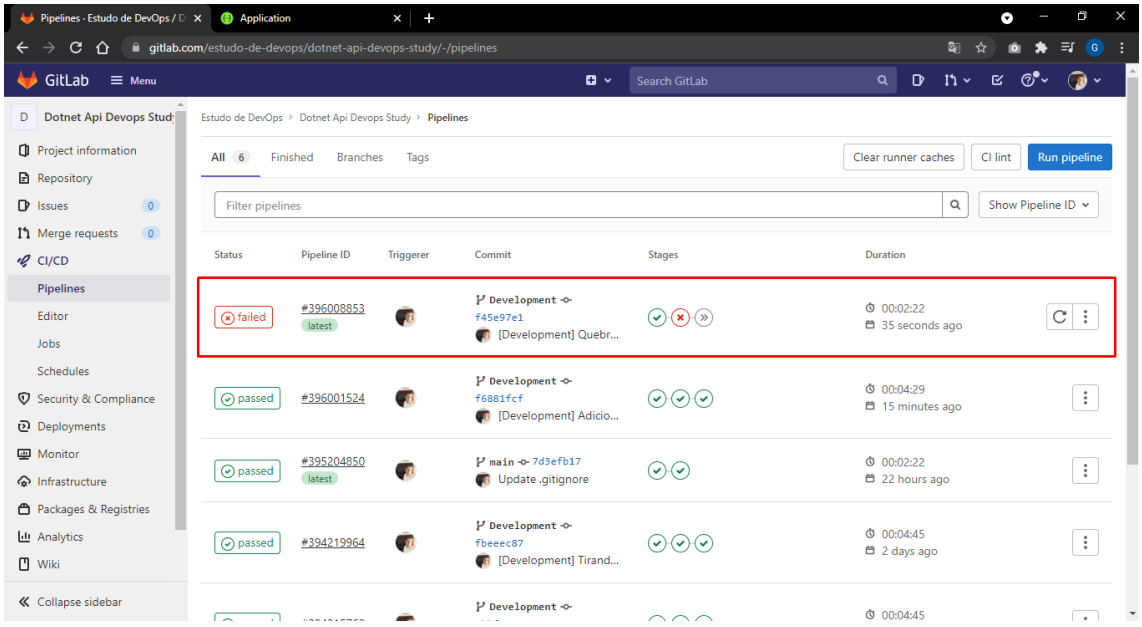
Figura 35: Teste unitário com erro proposital



Fonte: Autores

Após o *commit*, a etapa de teste na *pipeline* vai falhar (Figura 36).

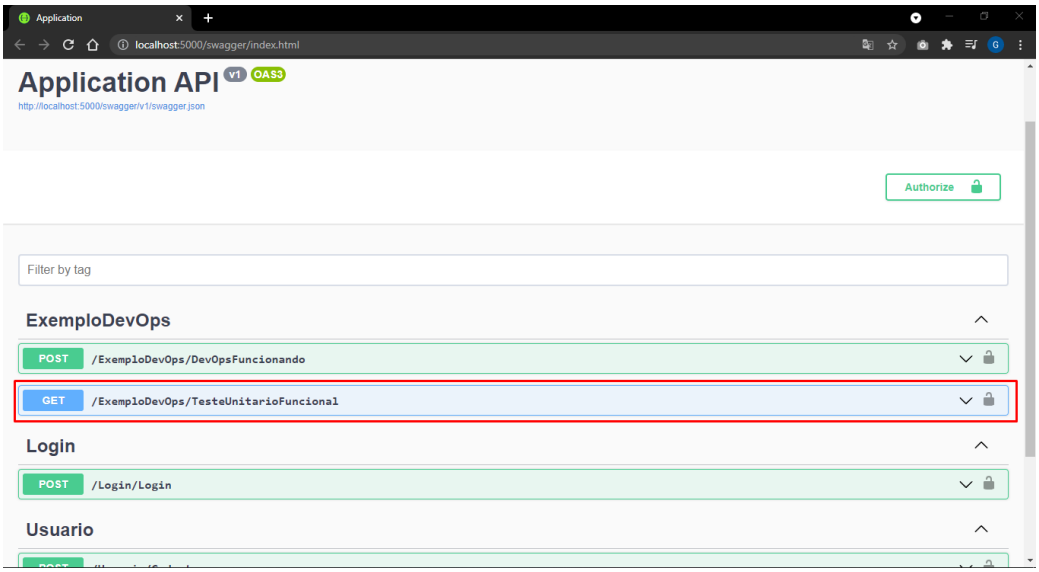
Figura 36: Erro na etapa de teste



Fonte: Autores

Vamos adicionar outro *endpoint* no *controller* existente para ver como o servidor refletirá, mantendo os testes unitários com falha. Testando localmente, o *endpoint* já está disponível (Figura 37).

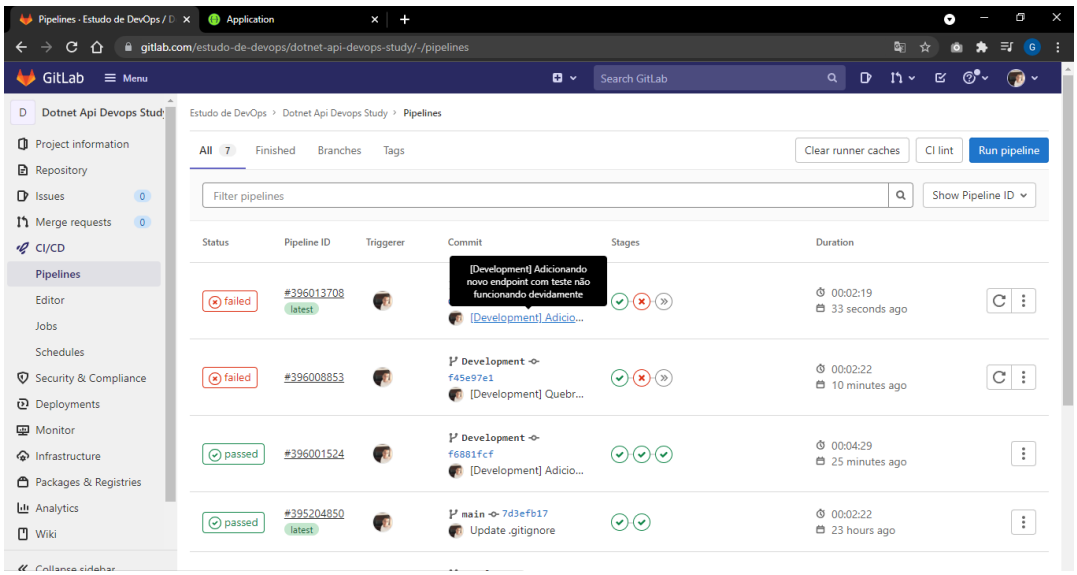
Figura 37: Endpoint novo criado para teste da pipeline com teste unitário “danificado”



Fonte: Autores

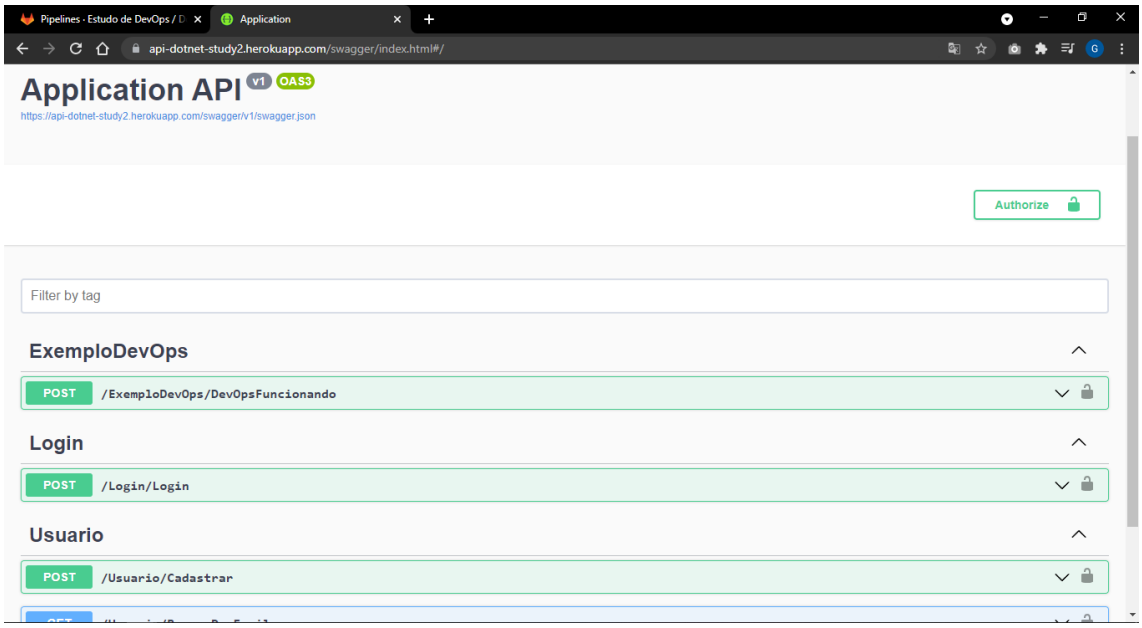
Após realizar o *commit*, a pipeline executará novamente e apontar um erro (imagem 37), consequentemente o servidor não terá o *endpoint* novo (Figura 39).

Figura 38: Erro na etapa de teste após *commit* de endpoint novo



Fonte: Autores

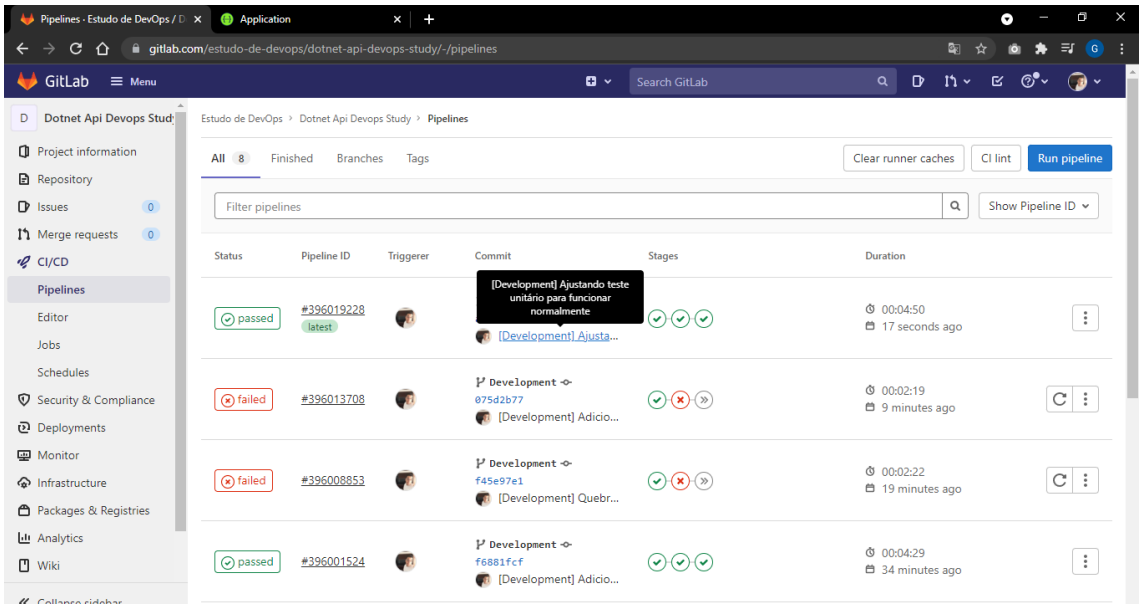
Figura 39: Endpoints no servidor pós falha na etapa de teste



Fonte: Autores

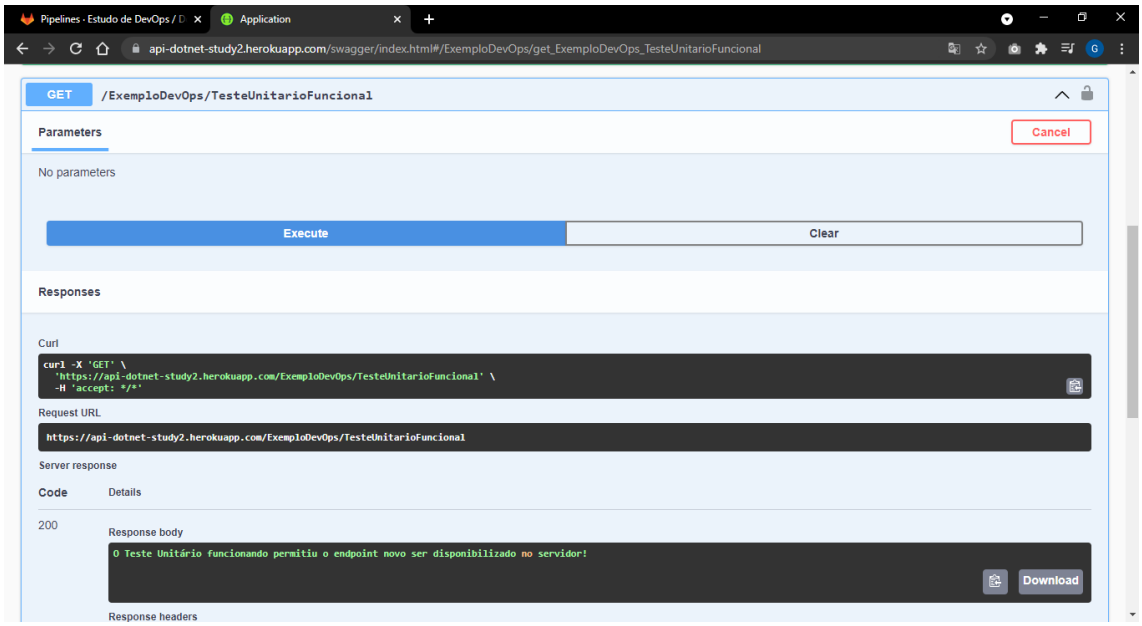
Agora, ao voltar o teste para o normal, a *pipeline* será executada com sucesso (Figura 39) e ao atualizar o site, o servidor terá o novo *endpoint* (Figura 41).

Figura 40: Sucesso em todas etapas após correção do teste unitário



Fonte: Autores

Figura 41: Resultado do endpoint no servidor após sucesso na pipeline



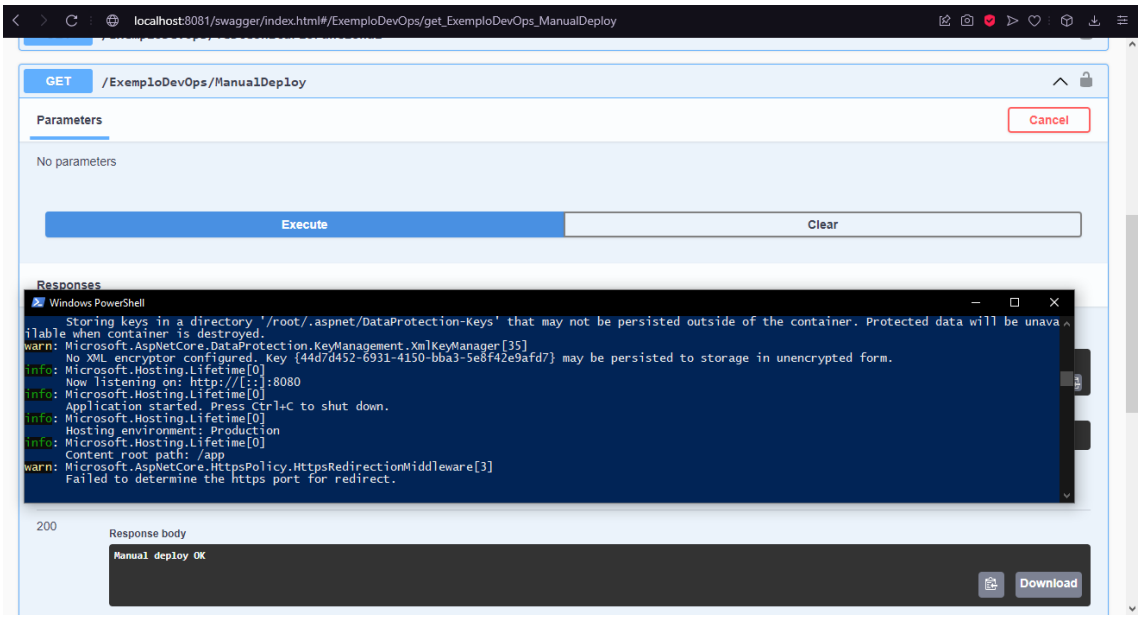
Fonte: Autores

3.3 Execução da proposta

Para medir as vantagens, é preciso medir o tempo de um *deploy* manual e os recursos braçais utilizados, para isso fora usado um cronometro medindo o tempo dos passos, utilizando uma máquina com as seguintes configurações: Placa Mãe com soquete AM4, Processador com 8 núcleos e 16 threads, 32GB (4x8 com 3000mhz cada) de Memória RAM DDR4.

Primeiro é necessário fazer o *build* da aplicação manualmente e executar os testes unitários. Logo em seguida precisa-se construir o container Docker que conterá nossa aplicação, como estamos um ambiente Windows para estudo, é essencial o Docker *desktop* instalado, para ser mais dinâmico, partiremos do pressuposto que já esteja instalado. Após construir a imagem e executar ela localmente para teste, percebe-se que a mesma já está funcionando (Figura 42).

Figura 42: Container em funcionamento local com o novo endpoint



Fonte: Autores

Para conseguir realizar o *deploy* da imagem Docker no Heroku, como é uma imagem já existente, precisamos apenas utilizar os seguintes comandos:

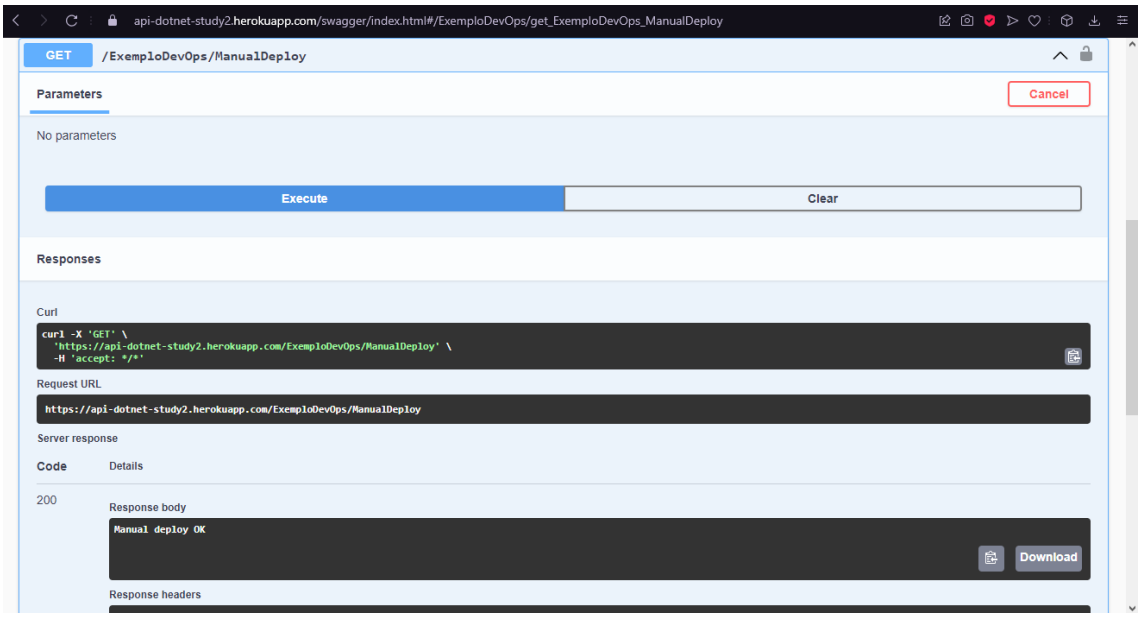
1 - docker tag (nomeDaSuaImagemDocker)
registry.heroku.com/(nomeDaSuaAplicacaoNoHeroku)/web
2 - docker push registry.heroku.com/(nomeDaSuaAplicacaoNoHeroku)/web

E em seguida utiliza-se o seguinte comando:

1 - heroku container:release web --app=(nomeDaSuaAplicacaoNoHeroku)

Após executar os comandos e acessar o endereço da aplicação, percebemos que o novo *endpoint* já está disponibilizado (Figura 43).

Figura 43: Container em funcionamento local com o novo endpoint



Fonte: Autores

Nessa etapa não foi levado em consideração o tempo de aprendizagem para o *deploy* manual, pois pode variar de acordo com o conhecimento e experiencia do desenvolvedor em questão. Porém este não deixa de ser um ponto importante do resultado pois, a aplicação tem o mesmo padrão de execução sempre, podendo ser utilizada por membros novos ou experientes do time, reduzindo muito o tempo de aprendizagem necessário sendo que, quando aplicado para um ambiente mais robusto, onde ocorre maior rotatividade de membros da equipe o resultado pode ser ainda mais expressivo.

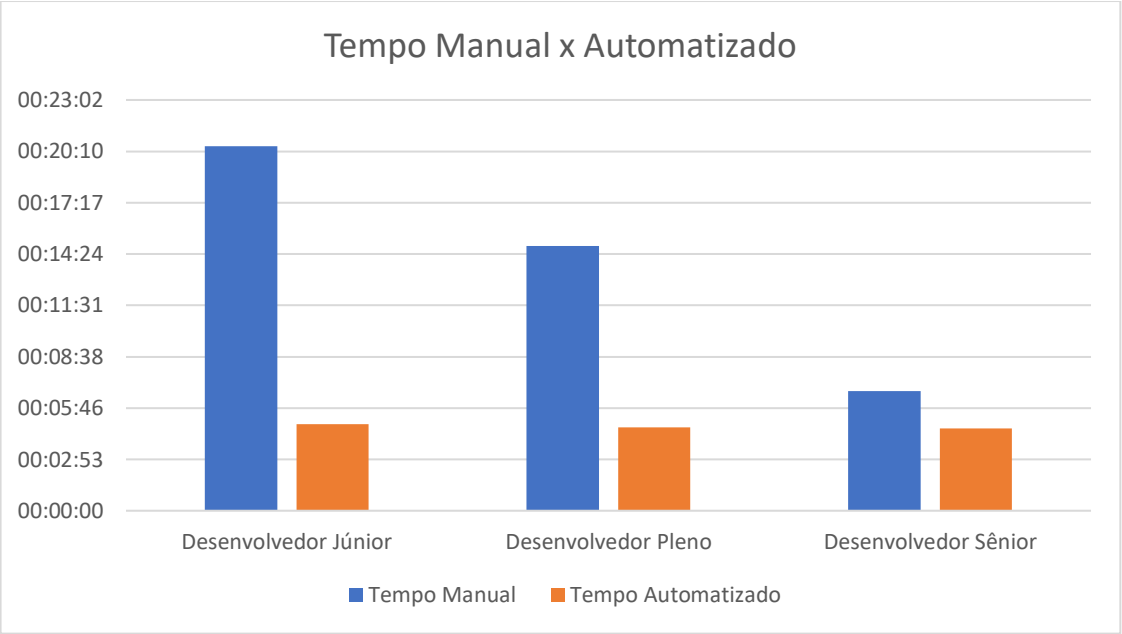
O teste explicado no início da secção foi executado com três desenvolvedores que possuem cargos de diferentes níveis de senioridade em suas respectivas empresas, assim podemos entender melhor não somente a diferença entre o tempo manual e automatizado, mas também o impacto que o conhecimento tem em cada modo, como pode ser visto nas figuras abaixo.

Figura 44: Tabela comparativa de resultados

Desenvolvedores	Tempo Manual	Tempo Automatizado
Desenvolvedor Júnior	20m 27s	4m 51s
Desenvolvedor Pleno	14m 51s	4m 40s
Desenvolvedor Sênior	6m 42s	4m 37s

Fonte: Autores

Figura 45: Gráfico comparativo de resultados



Fonte: Autores

Mesmo que o valor de tempo ganho não pareça ser tão expressivo, é importante lembrar que foi necessário designar um membro da equipe para que o teste manual fosse executado, este que poderia estar atuando em outra demanda.

Outro ponto importante é o impacto que o conhecimento e experiencia do desenvolvedor teve na execução dos cenários, na forma manual quanto menor a senioridade do desenvolvedor, maior era o tempo para execução além de alguns erros acontecerem no processo, por outro lado ao executar o processo criado, o resultado se mantem basicamente o mesmo independente do conhecimento de quem o executa.

4 CONCLUSÃO

Após a aplicação do trabalho ficou perceptível o impacto da aplicação de um processo estruturado utilizando práticas DevOps, mesmo contemplando apenas as funções base da gama de práticas contidas na metodologia, o processo criado mostra-se efetivo no que se propõe, reduzindo o tempo de *deploy*, desvinculando o processo do conhecimento de quem o executa e consolidando a base necessária para uma gama de automações possíveis.

No que diz respeito ao tempo de *deploy*, a redução foi em média 68% quando utilizada a forma automatizada, isso em um cenário de apenas um requisito para um desenvolvedor, aumentando a quantidade de ambos os fatores, estimando-se que o tempo do *deploy* manual deve sofrer aumento também. Desvincular o processo da ação de um ou mais desenvolvedores tem impacto direto na redução de tempo, pois não há mais dependência da disponibilidade do mesmo ou do conhecimento que ele possua.

Na questão conhecimento, o resultado se mostrou igualmente relevante, quanto menor a senioridade mais tempo o desenvolvedor levou para executar as tarefas de forma manual, por outro lado o modelo automatizado mantém seu resultado independente deste fato, fazendo com que não haja mais uma dependência de conhecimento para agilidade na execução de tarefas. Se levarmos em conta a rotatividade da área de tecnologia, faz-se extremamente necessário a remoção deste vínculo, uma vez que o desenvolvedor pode facilmente sair da empresa e levar o conhecimento consigo. Outro ponto importante é na questão dos testes, onde o conhecimento, disponibilidade e atenção do responsável tem impacto direto na qualidade do resultado, com este processo é preciso apenas vincular a execução dos testes ao processo de *deploy* e ele é feito de forma automatizada, sendo assim, há uma redução ainda maior no tempo de execução e um aumento na segurança e qualidade da funcionalidade a ser entregue.

Por fim, é importante ressaltar que este modelo de processo é uma versão mínima utilizável para comprovar o impacto a importância das práticas DevOps, ainda há muitas outras práticas que podem ser somadas ou processo, abrindo um leque de possibilidade de aplicação nas diversas realidades de empresas de software, todavia o processo aqui estruturado serve como base para todas essas aplicações e pode ser facilmente adaptado e melhorado de acordo com a necessidade e realidade do ambiente.

4.1 Trabalhos futuros

Para trabalhos futuros é fundamental o estudo de modelos e processos de testes automatizados, visando complementar com a área de testes e qualidade o modelo mínimo de estrutura criado neste trabalho. Ao criar um modelo base validado e abrangente, pode-se aplicar diversas técnicas e processos visando validar a eficácia do mesmo e integrá-lo ao processo completo fazendo com que os testes possuam uma estrutura mais sólida, tendo em vista que esse não era o foco principal do trabalho.

Outro fator importante é a aplicação em cenários maiores e mais complexos, com o objetivo de mensurar o ganho da utilização do processo nestes ambientes, permitindo que haja um levantamento de pontos que eventualmente precisem de otimizações.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILAR, E. **O que é e como funciona uma fábrica de software**. Profissionais TI, 2011. Disponível em: <https://www.profissionaisti.com.br/o-que-e-e-como-funciona-uma-fabricade-software/> Acesso em: 15 de abril de 2021.

AMARAL, A. **O que é infraestrutura como código?**. Tivit, 2018. Disponível em: <https://blog.tivit.com/o-que-e-infraestrutura-como-codigo> Acesso em: 10 de maio de 2021.

AUGUSTO, C. **Git, GitHub e GitLab o que é cada um deles?** Diolinux, 2020. Disponível em: <https://diolinux.com.br/editorial/git-github-e-gitlab.html> Acesso em: 10 de outubro de 2021.

BARBOSA, D. **Importância de DevOps para as organizações.** CEDRO, 2019. Disponível em: <https://blog.cedrotech.com/importancia-de-devops-para-as-organizacoes/> Acesso em: 06 de maio de 2021.

BATAGINI, R. **Como versionar utilizando GIT.** Biblioteca dos Devs, 2020. Disponível em <https://medium.com/biblioteca-dos-devs/como-versionar-utilizando-o-git-1f5d8fe2afcd> Acesso em: 10 de outubro de 2021.

BERTOLA, F. **Git, GitHub e GitLab: O que são e principais diferenças.** Zup, 2019. Disponível em: <https://www.zup.com.br/blog/git-github-e-gitlab> Acesso em: 10 de outubro de 2021.

CAROLI, P. MVP: **Conheça e saiba como usar o produto mínimo viável**, 2021. Disponível em: < https://www.caroli.org/mvp-produto-minimo-viavel/> Acesso em: 24 de outubro de 2021.

CRUZ, D. S. **3 problemas que uma cultura organizacional bem definida pode resolver em sua empresa**. Santo Caos, 2019. Disponível em: <https://www.santocaos.com.br/post/3problemas-que-uma-cultura-organizacional-bem-definida-pode-resolver-em-sua-empresa> Acesso em: 21 de abril de 2021.

DALMAZO, B. **Vantagens de utilização do Docker Container**. GlobalMind, 2021. Disponível em: <https://www.globalmind.com.br/vantagens-da-utilizacao-do-docker-container/> Acesso em: 24 de outubro de 2021.

DIAS, G. **Adoção da cultura DevOps: O que ainda falta?**. Baguete, 2018. Disponível em: <https://www.baguete.com.br/noticias/08/08/2018/adocao-da-cultura-devops-o-que-aindafalta.> Acesso em: 15 de abril de 2021.

EVEO, R. **Por que o versionamento de software é tão importante? Descubra!** EVEO, 2020. Disponível em: <https://www.eveo.com.br/blog/versionamento-de-software/> Acesso em: 10 de outubro de 2021.

GOMES, P. C. T. **Afinal, o que é Docker?** OPServices, 2018. Disponível em: <https://www.opservices.com.br/o-que-e-docker/> Acesso em: 24 de outubro de 2021.

GUEDES, M. **No final das contas: o que é o Docker e como ele funciona?** Treinaweb, 2018. Disponível em: <https://www.treinaweb.com.br/blog/no-final-das-contas-o-que-e-o-docker-e-como-ele-funciona> Acesso em: 24 de outubro de 2021.

GUERRERO, J. **Tendencias en DevOps: un mapeo sistemático de la literatura.** Universidad del Cauca, 2020. Disponível em: <https://www.researchgate.net/profile/CesarCalvache/publication/344478031_Tendencias_en_DevOps_un_mapeo_sistematico_de_la_literatura/links/5f7b493aa6fdcc0086576b2b/Tendencias-en-DevOps-un-mapeo-sistematico-dela-literatura.pdf.> Acesso em: 14 de abril de 2021.

JOUKOVSKI, C. V. R. **O que é Laravel? Conheça o framework de PHP mais utilizado**. Tecmundo, 2021. Disponível em <https://www.tecmundo.com.br/software/223718-laravel-conheca-o-framework-php-utilizado.htm> Acesso em: 25 de setembro de 2021.

LEE, T. G. **Bem-vindo ao IDE do Visual Studio. Microsoft**, 2021. Disponível em: <https://docs.microsoft.com/pt-br/visualstudio/get-started/visual-studio-ide?view=vs-2019> Acesso em: 24 de outubro de 2021.

MATSUMOTA, L. **Cultura DevOps na sua empresa**. Digital Strategy and IT Innovation, 2018. Disponível em: <https://leonardo-matsumota.com/2018/10/14/cultura-devops-na-suaempresa/>

NUNES, P. **Entenda por que versionamento de software é tão importante.** Disponível em: < https://gaea.com.br/entenda-por-que-versionamento-de-software-e-tao-importante/> Acesso em: 24 de outubro de 2021.

ONE, H. **O que é GIT e como ele otimiza seu desenvolvimento.** HostOne, 2019. Disponível em: <https://blog.hostone.com.br/o-que-e-git-e-como-ele-otimiza-seu-desenvolvimento/> Acesso em: 10 de outubro de 2021.

PRADO, S. **O que é um container linux?**. Sergio Prado Org, 2019. Disponível em: https://sergioprado.org/o-que-e-um-container-linux/

PRESSMAN, R. **Engenharia de Software: Uma Abordagem Profissional.** 7ª Ed., São Paulo: McGraw-Hill, 2011.

RAVOOF, S. **GitLab vc GitHub: Explore suas Principais Diferenças e Semelhanças.** Kinsta, 2021. Disponível em <https://kinsta.com/pt/blog/gitlab-vs-github/> Acesso em: 10 de outubro de 2021.

ROVEDA, U. **O que é framework, para que serve, vantagens e desvantagens**. Kenzie, 2021. Disponível em <https://kenzie.com.br/blog/framework/> Acesso em: 25 de setembro de 2021.

RAMOS, R. **Treinamento Prático em UML**. São Paulo: Digerati Books, 2006.

ROVEDA, U. **O que é Git e GitHub, como usar e quais são as vantagens?** Kenzie, 2021. Disponível em: <https://kenzie.com.br/blog/o-que-e-git/> Acesso em: 10 de outubro de 2021.

SACOLICK, I. **Práticas recomendadas de Devops: 5 métodos que você deve adotar**. CIO, 218. Disponível em: <https://cio.com.br/gestao/praticas-recomendadas-de-devops-5-metodosque-voce-deve-adotar/> Acesso em: 15 de abril de 2021.

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

SOUSA, L. **DevOps - Estudo de Caso**. ISCAC, 2019. Disponível em: <https://comum.rcaap.pt/handle/10400.26/31932>

SOMMERVILLE, I. **Software engineering**. 7th. ed. Addison-Wesley, 2004.

SHAPIRO, J. **O que é controle de versão**. Atlassian, 2020. Disponível em: <https://www.atlassian.com/br/git/tutorials/what-is-version-control#benefits-of-versioncontrol>

VIRMANI, M. **Understanding Devops & Bridging The Gap From Continuous Integration To Continuous Delivery**. Fifth International Conference on the Innovative Computing Technology (INTECH 2015). IEEE, 78–82. New York, NY, USA: IEEE.

ZUCHER, V. **O que é um framework? Pra que serve e por que você deveria saber?** Le Wagon, 2020. Disponível em <https://www.lewagon.com/pt-BR/blog/o-que-e-framework> Acesso em: 25 de setembro de 2021.