Integração Contínua para Automação de Testes Utilizando Cypress, Docker e Jenkins

Lucas Rosa de Melo, Jaqson Dalbosco

Curso de Ciência da Computação – Universidade de Passo Fundo BR 285 Km 292,7 – Bairro São José – Passo Fundo, RS, Brasil

{175801, jaqson}@upf.br

Abstract. This work explores the integration of three essential tools in software development and testing: Cypress, Docker, and Jenkins, aimed at end-to-end test automation in a DevOps environment. The main objective was to create an encapsulated test system for continuous execution in an automated pipeline, reducing the dependency on specific technical knowledge and promoting reproducibility and scalability of API tests. Cypress was used for test automation, Docker for environment encapsulation, and Jenkins for creating a continuous integration pipeline that automates test execution. The implementation of this system proved effective in optimizing the efficiency and consistency of validation processes, presenting a practical solution for development and quality teams seeking greater agility and accuracy in software testing.

Resumo. Este trabalho explora a integração de três ferramentas essenciais no desenvolvimento e testes de software: Cypress, Docker e Jenkins, visando a automação de testes end-to-end em um ambiente DevOps. O objetivo principal foi criar um sistema de testes encapsulado para execução contínua em uma pipeline automatizada, o que reduz a dependência de conhecimento técnico específico e promove a reprodutibilidade e a escalabilidade dos testes de API. Para isso, foi utilizado o Cypress para a automação dos cenários de testes, Docker para encapsulamento do ambiente e Jenkins para a criação de uma pipeline de integração contínua que automatiza a execução dos testes. A implementação desse sistema mostrou-se eficaz para otimizar a eficiência e a consistência dos processos de validação, demonstrando uma solução prática para equipes de desenvolvimento e qualidade que buscam maior agilidade e precisão nos testes de software.

1. Introdução

A área de engenharia de software desempenha um papel fundamental na melhoria contínua dos processos de desenvolvimento de software, buscando constantemente formas de melhorar a eficiência, a qualidade e a colaboração entre as equipes de desenvolvimento. Além disso, a área é amplamente rica em cultura, dentre essas culturas pode-se citar a metodologia *DevOps* que integra desenvolvimento de software (Dev) e operações de TI (Ops), promovendo colaboração e comunicação entre essas equipes tradicionalmente separadas. A implementação do *DevOps* visa acelerar o ciclo de vida do desenvolvimento, permitindo entregas mais rápidas e frequentes sem comprometer a qualidade. Isso é alcançado através da automação de processos,

integração contínua e monitoramento constante, o que reduz erros e aumenta a eficiência operacional. [Fastercapital, 2023]

Com o passar do tempo, *DevOps* passou a englobar muito mais processos, dentre eles todo o ciclo de testes que anteriormente eram realizados por equipes de QA (*Quality Assurance*) especializadas e de forma manual. Ao englobar isso surge a oportunidade da implementação de tecnologias mais avançadas para testes. A automação de testes *end-to-end* tornou-se uma prática importante para garantir a qualidade do software, detectando precocemente possíveis falhas e acelerando os ciclos de desenvolvimento. Docker (plataforma de contêiner) e Jenkins (servidor de automação) tornam-se ferramentas essenciais para encontrar um ambiente de teste consistente, escalável e eficiente.

Metodologias ágeis no âmbito de desenvolvimento de software exigem cada vez mais que a infraestrutura que mantém os serviços seja entregue de forma rápida e eficaz, de acordo com Portes (2022). A necessidade de ferramentas como o Docker decorre da complexidade inerente de utilizar vários ambientes durante o desenvolvimento, teste e implantação. Programadores podem enfrentar desafios ao transferir aplicações entre sistemas diferentes, resultando em várias discrepâncias. Com Docker, pode-se encapsular o serviço com todas as suas dependências em um único ambiente chamado de contêiner. À medida que o Docker se estabelece como uma solução vital no desenvolvimento de software, sua integração com ferramentas de automação, como o Jenkins, torna-se uma estratégia poderosa para aprimorar ainda mais a eficiência no ciclo de vida do desenvolvimento. A ideia de algo junto (integrado) e feito sem parar (continuamente) é o que traz a ideia de IC (Integração Contínua) em DevOps (BOAGLIO, 2016) e graças ao Jenkins esse tipo de estado pode ser alcançado. Com o uso do Cypress como ferramenta de testes *end-to-end* em um ambiente de Integração Contínua já definido, reprodutibilidade, consistência e eficiência são resultados obtidos.

Segundo Ferreira (2010), muitos projetos são entregues com atraso devido à falta de um desempenho satisfatório em processos de desenvolvimento e testes, no quesito de qualidade, tarefas manuais como a configuração de um projeto automatizado de testes ponta a ponta, onde cada dependência precisa estar funcionando corretamente, ou então a forma como os cenários serão executados, levam tempo, o'que torna necessário a atenção constante de um profissional de qualidade para esse tipo de situação. Caso um membro do time de desenvolvimento precise de um relatório com resultados de teste, ele dependerá do profissional para extrair essas informações, ou então, deverá configurar em sua própria máquina todo o projeto de teste e encarar a curva de aprendizado que aquela nova tecnologia irá exigir, consumindo mais ainda o seu tempo.

Com isso em mente, esse trabalho tem como objetivo apresentar uma solução capaz de facilitar etapas de testes de software utilizando as ferramentas já apresentadas, visando criar um projeto de testes de sistema encapsulado para a execução continua em uma pipeline automatizada, de forma que, para a execução, não demande nenhum conhecimento na tecnologia usada para desenvolver aqueles cenários de teste, demonstrando as vantagens da prática de integração das tecnologias Cypress, Docker e Jenkins para times de desenvolvimento e qualidade de software no que diz respeito a eficiência e escalabilidade.

O artigo será organizado da seguinte forma: a seção 2 irá apresentar uma revisão sobre a área de teste de software, suas vantagens, importância e algumas referências importantes. A seção 3 será dedicada a trabalhos relacionados, trabalhos esses que se assemelha ao apresentado neste artigo. A seção 4 será composta pelas especificações do trabalho. A seção 5 demonstrará como foi todo o processo de implementação e a utilização das ferramentas necessárias. E por fim, na seção 6 será apresentado os resultados obtidos após a validação.

2. Revisão sobre testes

Testes de software são processos fundamentais no ciclo de desenvolvimento de sistemas, cuja finalidade é garantir que um programa funcione conforme o esperado, atendendo aos requisitos especificados e operando com qualidade e eficiência. Esses testes visam identificar possíveis erros, defeitos ou falhas que possam comprometer a funcionalidade e a usabilidade do software, assegurando que o produto final entregue ao usuário esteja livre de falhas que impactem sua experiência ou a segurança dos dados manipulados.

Além de assegurar a funcionalidade, os testes de software exercem um papel essencial na melhoria contínua do produto. Por meio deles, é possível realizar ajustes finos que otimizam o desempenho e identificam áreas para aprimoramento, garantindo que o software não apenas funcione, mas também opere com máxima eficiência. Em ambientes de desenvolvimento ágil, onde o ritmo de entrega de novas versões e funcionalidades é acelerado, os testes tornam-se ainda mais cruciais, pois permitem uma validação constante do sistema, facilitando a rápida detecção de problemas e a manutenção da estabilidade do software.

Segundo Benitti (2014), existem diferentes tipos de testes de software, cada um com objetivos específicos e aplicados em diferentes camadas do desenvolvimento para assegurar a funcionalidade, qualidade e desempenho do sistema. Para estruturar essa diversidade de abordagens, é comum adotar o conceito de pirâmide de testes, que organiza os tipos de testes em camadas, proporcionando uma visão estratégica de como cada teste deve ser utilizado ao longo do ciclo de desenvolvimento. A base da pirâmide representa os testes que devem ser mais frequentes e automatizados, enquanto o topo, que contém os testes menos numerosos e mais complexos, indica aqueles que podem ser mais dispendiosos e menos frequentes.

Na base da pirâmide, encontram-se os testes unitários, que têm como objetivo verificar se unidades individuais de código funcionam corretamente. Esses testes são rápidos, de baixo custo e altamente automatizados, sendo essenciais para garantir que cada parte do sistema funcione conforme o esperado isoladamente.

Subindo na pirâmide, encontram-se os testes de integração, que são projetados para avaliar a interação entre diferentes componentes ou módulos do sistema. Em muitos projetos, essa camada inclui os testes de API [Rolt, 2024]. Testes de API verificam se as chamadas e respostas da API estão funcionando corretamente, validando o comportamento da interface, o processamento de dados e a compatibilidade com os clientes ou serviços que consomem essa API. Esses testes são importantes para detectar problemas de comunicação entre diferentes partes do sistema, garantindo que os módulos integrados funcionem de forma coesa e confiável. Esses testes cobrem uma ampla gama de verificações, como a correta resposta aos parâmetros enviados, a

performance e o tempo de resposta, e o manuseio de erros ou dados incorretos. Com o uso de ferramentas de automação, é possível assegurar que a API opere de maneira estável e segura, proporcionando uma base sólida para as demais integrações do sistema.

No topo da pirâmide de testes estão os testes de interface de usuário (ou testes de sistema), que avaliam o sistema como um todo em um cenário o mais próximo possível do real. Embora sejam menos frequentes e mais dispendiosos para automatizar, esses testes têm grande importância para verificar a experiência do usuário e a usabilidade da aplicação [Rolt, 2024].

Além disso, existem algumas classificações para as técnicas de testes, sendo elas caixa-branca e caixa-preta. Nos testes caixa-preta, a ênfase está em avaliar a funcionalidade do software sem conhecimento de seu código ou estrutura interna. O testador verifica as entradas e saídas, validando se o software responde conforme esperado aos requisitos especificados. Essa abordagem é útil para identificar problemas de usabilidade e verificar se o sistema atende às expectativas do usuário final. Por outro lado, os testes caixa-branca envolvem o conhecimento detalhado do código e da lógica interna do software. Aqui, o testador examina o funcionamento interno do sistema, verificando cada caminho de execução, estrutura condicional e laços para garantir que todas as partes do código sejam executadas corretamente. Essa abordagem ajuda a identificar falhas na lógica do programa e a melhorar a cobertura de código [Valente, 2020].

A automação de testes é o processo de utilizar ferramentas e scripts para realizar testes de software de maneira automática, sem a necessidade de intervenção manual constante. Essa prática visa aumentar a eficiência, precisão e cobertura dos testes, especialmente em projetos onde o desenvolvimento é contínuo e exige validações frequentes, como nos métodos ágeis e DevOps.

Ao automatizar os testes, é possível executar cenários repetitivos e complexos de forma rápida e confiável, o que libera a equipe de qualidade para se concentrar em análises mais estratégicas e menos mecânicas. A automação é particularmente útil em testes de regressão, que precisam ser executados a cada nova modificação no código, garantindo que novas funcionalidades ou correções não introduzem erros em funcionalidades existentes.

As ferramentas de automação de testes, como Selenium, Cypress, JUnit e outras, permitem que as equipes configurem e executem testes em diferentes camadas do sistema, desde testes unitários até testes de API e de interface. Além disso, a automação facilita a implementação de pipelines de integração contínua e entrega contínua.

3. Trabalhos Relacionados

Nos últimos anos, a aplicação de práticas de DevOps e automação de testes tem crescido consideravelmente, permitindo que equipes de desenvolvimento entreguem software de maneira mais eficiente e com maior confiabilidade. Estudos recentes investigam diversas abordagens para configurar pipelines de DevOps e automação de testes, focando em ferramentas que permitem gerenciar e automatizar processos complexos de desenvolvimento, teste e implantação. Essas pesquisas trazem contribuições significativas para a compreensão e implementação de ambientes de desenvolvimento modernos e são essenciais para a fundamentação deste trabalho. Abaixo, são discutidos

dois estudos relevantes que exploram o uso de Docker e Cypress para automatizar processos no ciclo de vida de desenvolvimento de software.

3.1 DevOps Pipeline with Docker

Mironov (2018), apresenta um estudo sobre a implementação de um pipeline DevOps utilizando Docker como tecnologia central. Nesse estudo, o autor explora a criação de um modelo reutilizável de pipeline automatizado, capaz de suportar o desenvolvimento de software moderno com alta confiabilidade e escalabilidade. Docker é integrado ao pipeline para gerenciar o desenvolvimento, testes e implantação de aplicações, promovendo máxima automação e garantindo transparência nos processos. Esse estudo foca na configuração de ambientes de desenvolvimento até a implantação em produção, abordando também o uso de orquestração de contêineres com Rancher, uma ferramenta utilizada para gerenciar de forma eficiente os contêineres distribuídos.

A abordagem de Mironov se concentra na integração de ferramentas como Docker e Jenkins para suportar práticas modernas de DevOps. No trabalho, Jenkins é utilizado como servidor de automação, configurando e executando pipelines de integração contínua (CI) e entrega contínua (CD) com base nas imagens de contêineres geradas pelo Docker. A aplicação de referência usada no pipeline é uma aplicação web simples, chamada Voting App, que envolve a criação de contêineres tanto para o frontend quanto para o backend da aplicação, demonstrando o processo de dockerização e execução de testes automatizados. Outro ponto de destaque no trabalho de Mironov, é também relevante para o contexto deste artigo, é a utilização de Docker Compose para o desenvolvimento local. Docker Compose permite a criação de ambientes consistentes e totalmente automatizados, orquestrando múltiplos contêineres localmente. Isso simplifica o processo de desenvolvimento e testes ao facilitar a gestão dos serviços que compõem a aplicação. Assim como no presente estudo, Docker Compose foi uma ferramenta fundamental para a execução e testes de aplicações em um ambiente de desenvolvimento local antes da integração com o pipeline de CI/CD.

3.2 Modern Web Automation with Cypress.Io

Thekkan e Anuar (2022) apresentam um estudo sobre a implementação de automação de testes utilizando a ferramenta Cypress.io em um ambiente de Integração Contínua (CI). O trabalho aborda a criação de um framework de automação de testes para um projeto em desenvolvimento e destaca a importância de uma solução mais rápida e eficiente para lidar com as demandas das aplicações web modernas. Nesse estudo, foi implementado um ciclo ágil de desenvolvimento, integrando o Cypress ao Jenkins para garantir a automação dos testes de ponta a ponta e a execução contínua dentro do pipeline de CI .

O estudo de Thekkan e Anuar destaca o uso de Cypress como uma ferramenta de automação que resolve muitos dos problemas com ferramentas tradicionais de teste, como tempos de espera inconsistentes e dificuldades de configuração. Cypress se diferencia por executar os testes diretamente no navegador, capturando eventos em tempo real e oferecendo feedback mais rápido e confiável. Para maximizar a eficiência, a integração com Jenkins permite que os testes sejam executados automaticamente após

cada modificação no código, garantindo uma verificação contínua e estável da qualidade do software [Thekkan e Anuar 2022].

Outro ponto de relevância no estudo é a estrutura modular adotada no framework de testes, o que facilita a manutenção e o reaproveitamento de código. O uso do Page Object Model (POM) no Cypress simplifica a separação entre páginas e testes, tornando a automação mais robusta e escalável. Esse enfoque também é implementado no presente estudo, que explora a integração de Cypress e Jenkins em um pipeline contínuo, reforçando a aplicabilidade das melhores práticas de automação no desenvolvimento de software ágil.

4. Especificações para o projeto

O projeto desenvolvido consiste em um sistema de testes encapsulado que possibilita a execução contínua em uma pipeline automatizada. O objetivo central é demonstrar as vantagens da integração de tecnologias como Cypress, Docker e Jenkins para equipes de desenvolvimento e qualidade de software, com ênfase na melhoria de eficiência e escalabilidade dos testes de sistema. Este projeto foi estruturado para suportar práticas de Integração Contínua (CI), promovendo uma automação robusta que facilita a execução e o monitoramento dos testes de forma repetida e consistente.

Esta seção apresenta as especificações detalhadas do projeto, incluindo o aplicativo que serviu como base de desenvolvimento e os cenários de teste aplicados para validar a API. Primeiramente, será descrito o aplicativo original, suas funcionalidades e seu papel na conscientização e prevenção do câncer infantojuvenil. Em seguida, serão detalhados os cenários de teste projetados para assegurar a funcionalidade, segurança e precisão da API, que foram estruturados para garantir o correto processamento de sintomas e fatores de risco, conforme as diretrizes do projeto.

4.1 AppCancer

Neste projeto, foi utilizado como base um aplicativo previamente desenvolvido com o objetivo de apoiar a conscientização e prevenção do câncer infantojuvenil. Este aplicativo, criado para o Centro Oncológico Infantojuvenil do Hospital São Vicente de Paulo, é destinado a informar o público sobre sinais e sintomas associados aos diversos tipos de câncer que afetam crianças e adolescentes, promovendo o diagnóstico precoce. A ferramenta integra um algoritmo que avalia sintomas e fatores de risco, orientando o usuário a procurar aconselhamento médico caso sejam identificados sinais de alerta relevantes.

Com o intuito de facilitar o acesso a informações críticas sobre o câncer infantojuvenil, o aplicativo oferece uma interface acessível a pacientes, familiares e profissionais de saúde. Suas funcionalidades incluem a identificação de possíveis sintomas e uma base de dados informativa sobre os tipos de câncer, contribuindo assim para a redução de diagnósticos tardios e aumento da taxa de sobrevida. Essa estrutura foi fundamental como base para a criação do projeto de testes, pois permitiu avaliar a consistência dos processos de análise de sintomas e a precisão na exibição de informações, assegurando a qualidade e a confiabilidade dos resultados apresentados.

4.2 Cenários de teste

Os testes foram aplicados exclusivamente à API do aplicativo, garantindo que as funcionalidades do backend fossem verificadas independentemente da interface gráfica (frontend). Esse foco permitiu validar a robustez e precisão dos processos de análise de sintomas e de retorno de informações críticas para a experiência do usuário. Para a API, foram definidos cenários de testes que abrangem diferentes aspectos essenciais para o funcionamento do aplicativo, segue ilustrado na Figura 1, aspectos como o body da requisição e seu status. Esses cenários incluem: testes para garantir que apenas usuários autorizados possam acessar determinados recursos, verificando a segurança e a proteção dos dados, validação da integridade dos dados transmitidos entre o cliente eo servidor, assegurando que as informações enviadas estejam corretas e completas e cenários que verificam se a API realiza corretamente o processamento de sintomas informados pelos usuários e a associação com possíveis riscos, retornando orientações adequadas.

Cenário	Deve registrar user com sucesso
Method	POST
Host	**BASE_URL**/auth/register
Body	"login":"admin", "password":"8en4apEX2vpPyYM", "role":"ADMIN", "nome":"Ana"
Assertions	STATUS CODE : 200

Cenário	Não deve registrar user sem senha
Method	POST
Host	**BASE_URL**/auth/register
Body	"login":"admin", "password":"", "role":"ADMIN", "nome":"Ana"
Assertions	STATUS CODE : 400

Cenário	Deve logar com sucesso
Method	POST
Host	**BASE_URL**/auth/login
Body	"login": "admin", "password": "8en4apEX2vpPyYM"
Assertions	STATUS CODE : 200

Cenário	Deve registrar um grupo de risco com sucesso
Method	POST
Host	**BASE_URL**/grupoDeRisco
Body	"nome": "Grupo de Risco", "observacao": "Observação..."
Assertions	STATUS CODE : 200

Cenário	Não deve registrar um grupo de risco sem um nome
Method	POST
Host	**BASE_URL**/grupoDeRisco
Body	"nome": "", "observacao": "Observação..."
Assertions	STATUS CODE : 400

Cenário	Deve registrar um sintoma com sucesso
Method	POST
Host	**BASE_URL**/sintoma
Body	"nome" : "Sintoma", "descricao" :"Descrição..", "observacao" : "Observação..."
Assertions	STATUS CODE : 200

Cenário	Não deve registrar um sintoma sem o nome
Method	POST
Host	**BASE_URL**/sintoma
Body	"nome" : "", "descricao" :"Descrição..", "observacao" : "Observação..."
Assertions	STATUS CODE : 400

Cenário	Não deve registrar um paciente sem o seu nome
Method	POST
Host	**BASE_URL**/paciente
Body	"nome": "", "dataNascimento": "2023-03-30", "sexo": "M", "latitude": 34.1241, "longitude": -13.235, "sintomaPacienteDTOList": [{ "idSintoma": 1, "duracaoDias" : 3}], "grupoDeRiscoPacienteDTOList" : [{"idGrupoDeRisco" : 1}]
Assertions	STATUS CODE : 400

Cenário	Deve registrar um paciente com seu grupo de risco e sintoma com sucesso
Method	POST
Host	**BASE_URL**/paciente
Body	"nome": "Lucas", "dataNascimento": "2023-03-30", "sexo": "M", "latitude": 34.1241, "longitude": -13.235, "sintomaPacienteDTOList": [{ "idSintoma": 1, "duracaoDias" : 3}], "grupoDeRiscoPacienteDTOList" : [{"idGrupoDeRisco" : 1}]
Assertions	STATUS CODE : 200

Figura 1. Cenários de teste para a API

Fonte: Elaborada pelo autor

A inclusão de cenários detalhados, como a validação de autenticação, integridade de dados e processamento lógico, garante uma cobertura mais abrangente, assegurando que a API não apenas funcione conforme o esperado, mas também mantenha a segurança e a confiabilidade

5. Materiais e Métodos

Para assegurar uma estrutura robusta e confiável, foi necessário utilizar ferramentas e práticas de automação que permitissem a repetição e a validação contínua dos testes. A seleção cuidadosa dos materiais e métodos visou integrar tecnologias que, em conjunto, garantem portabilidade, escalabilidade e precisão no processo de testes de API. Esta seção descreve as ferramentas escolhidas e as etapas de desenvolvimento, detalhando cada aspecto envolvido na criação, configuração e execução dos testes automatizados. A seguir, são apresentados os principais recursos utilizados e o desenvolvimento do projeto.

5.1 Ferramentas

Foi utilizada uma combinação de ferramentas amplamente reconhecidas no mercado de testes de software e automação, incluindo Cypress, Docker e Jenkins. Cada uma dessas ferramentas desempenha um papel essencial, tanto para a automação de testes de API quanto para a configuração de um pipeline contínuo robusto e escalável.

5.1.1 Cypress

O Cypress foi utilizado para automatizar os testes dos cenários definidos para a API, garantindo uma verificação eficiente e precisa das funcionalidades do backend. Sendo uma ferramenta moderna para automação de testes end-to-end, o Cypress é amplamente valorizado pela facilidade de uso e pela capacidade de realizar testes de interface e API. Entre suas vantagens, destaca-se o feedback em tempo real sobre o status dos testes, facilitando a identificação e correção de falhas, especialmente útil em pipelines de CI/CD que dependem de respostas rápidas para manter a produtividade.

Além disso, o Cypress é executado diretamente no navegador, permitindo um controle preciso do DOM e proporcionando uma simulação realista das interações de API, o que torna o teste mais robusto e fidedigno. Sua API intuitiva e bem documentada o torna acessível tanto para desenvolvedores experientes quanto para iniciantes, permitindo a criação de testes consistentes de forma rápida e escalável. Por fim, sua execução em ambiente isolado e a fácil integração com CI/CD contribuem para a reprodutibilidade e confiabilidade dos testes, aspectos fundamentais para o sucesso do projeto.

5.1.2 Docker

Para assegurar a portabilidade e a consistência dos testes automatizados, foi utilizada a tecnologia Docker para encapsular todo o ambiente necessário para a execução do Cypress. O Docker permite que o ambiente de testes, incluindo todas as dependências,

configurações e scripts, seja empacotado em uma imagem Docker. Essa abordagem apresenta inúmeras vantagens que otimizam o processo de execução dos testes em diferentes máquinas e ambientes, eliminando a necessidade de configurações manuais adicionais.

Com o uso do Docker, o ambiente de testes torna-se completamente replicável, permitindo que qualquer pessoa, independentemente da configuração de sua máquina, possa executar os testes de maneira idêntica ao ambiente original. Isso é especialmente valioso em equipes de desenvolvimento distribuídas, onde diferenças de sistema operacional ou configurações locais podem impactar os resultados dos testes. A utilização de uma imagem Docker garante que todos os testes sejam realizados em um ambiente padronizado, minimizando erros decorrentes de inconsistências ambientais.

Além disso, o Docker simplifica o processo de distribuição do ambiente de testes. Uma vez que a imagem Docker esteja criada e armazenada em um repositório como o Docker Hub, qualquer membro da equipe ou servidor pode baixá-la e utilizá-la imediatamente, sem a necessidade de instalações complexas. Isso reduz significativamente a curva de aprendizado para novos desenvolvedores e testadores que precisem interagir com o sistema.

A consistência é outro ponto forte dessa ferramenta, pois Docker assegura que os testes sejam executados no mesmo ambiente em cada execução, eliminando problemas de incompatibilidade e garantindo resultados estáveis. O isolamento de ambiente, ao encapsular o projeto Cypress, evita conflitos com a configuração local e permite testes paralelos, o que otimiza o tempo de execução e reduz interferências. Além disso, Docker oferece uma integração perfeita com ferramentas de CI/CD, facilitando a construção e destruição automáticas de imagens, o que contribui para a eficiência e confiabilidade dos processos automatizados deste projeto.

5.1.3 Jenkins

Jenkins foi escolhido como a ferramenta de integração contínua para orquestrar o pipeline automatizado que gerencia o projeto. O Jenkins foi configurado para baixar e executar automaticamente a imagem Docker do Cypress, garantindo que os testes sejam executados de forma contínua e sem necessidade de intervenção manual. A automação completa do pipeline proporciona um fluxo de trabalho eficiente, com o Jenkins realizando desde a preparação do ambiente até a execução dos testes e a geração de relatórios.

Outra vantagem do Jenkins é sua compatibilidade com uma vasta gama de plugins, incluindo suporte para Docker e Cypress, o que facilita a construção de pipelines mais complexos e integrados. Além disso, o Jenkins permite agendar e monitorar a execução dos testes, oferecendo feedback rápido e auxiliando na resolução de problemas em tempo real. Sua arquitetura modular permite adaptações e escalabilidade, atendendo tanto a projetos pequenos quanto a grandes necessidades de CI/CD.

A combinação de Cypress, Docker e Jenkins oferece um ambiente de testes robusto e eficiente, onde o Cypress fornece uma estrutura de testes prática e eficaz, Docker assegura a consistência e portabilidade do ambiente encapsulado, e Jenkins possibilita a automação contínua e escalável dos processos de testes. Essa integração de

ferramentas facilita não apenas a implementação e execução dos testes, mas também promove uma cultura de desenvolvimento ágil e focada na qualidade, contribuindo para práticas modernas de integração e entrega contínuas.

5.2 Desenvolvimento da integração

O desenvolvimento da integração para o projeto foi dividido em várias etapas para assegurar uma organização eficaz. A primeira etapa consistiu em configurar o aplicativo *AppCancer* localmente, executando-o na máquina e verificando seu funcionamento. Com o aplicativo em execução, foi realizada uma análise dos principais endpoints disponíveis, observando-se as funcionalidades e os tipos de resposta que cada um deles retornava. Essa análise permitiu identificar e definir os cenários de teste necessários para avaliar o funcionamento da API.

5.2.1 Cypress com Node.js

Com os cenários de teste definidos, o desenvolvimento do projeto de testes automatizados foi iniciado utilizando a ferramenta Cypress, que é desenvolvida com Node.js, uma plataforma de desenvolvimento baseada na linguagem JavaScript que permite a execução de código JavaScript no servidor, o que é essencial para criar aplicações de alto desempenho e escaláveis.

Para rodar o Cypress com Node.js, foram instaladas algumas dependências, e o ambiente do projeto foi configurado para garantir a execução correta dos testes. Com essas configurações em ordem, para cada contexto do aplicativo, foi criado um arquivo contendo os cenários daquele contexto, por exemplo, na figura 2 o contexto em questão foi o de Grupo de Risco, dessa forma, ele é inserido em *Describe*, e dentro dos contextos são inseridos os cenários *its*, os cenários chamam a classe responsável pela requisição da API daquele contexto em específico, passando um *body,* no caso, *riskGroup.* Após isso, é obtida a response retornada pela API e validado o *Status Code.* Esses testes foram então executados localmente, interagindo diretamente com os endpoints da API do aplicativo.

Figura 2. Script com cenários de teste

Fonte: Elaborada pelo autor

Os cenários de teste automatizados deste projeto foram desenvolvidos seguindo um mesmo padrão. Para cada contexto testado, foram criados pelo menos dois cenários: um cenário de sucesso, em que a API deveria retornar a resposta esperada, e um cenário de falha, simulando condições que deveriam resultar em erros controlados ou respostas inválidas. Cada contexto foi encapsulado em seu próprio arquivo dentro da estrutura do projeto Cypress, o que facilitou a separação lógica e a manutenção dos testes. Essa abordagem modular permitiu que os testes fossem organizados de maneira clara, garantindo que alterações em um contexto não impactam os demais. Além disso, todas as informações necessárias para os bodys das requisições foram armazenadas em arquivos JSON separados. Essa prática não apenas padronizou os dados utilizados nos testes como também simplificou a reutilização e a atualização das informações, uma vez que qualquer modificação nos dados de entrada poderia ser feita diretamente nos arquivos JSON, sem a necessidade de alterar os scripts de teste.

5.2.2 Gitlab

Para gerenciar o código do projeto de testes, foi utilizado o GitLab, uma plataforma de versionamento e hospedagem de código que permite o controle e a organização eficiente do projeto. GitLab facilita o armazenamento seguro do código-fonte, além de oferecer funcionalidades avançadas de CI/CD, integração com repositórios e ferramentas de automação.

5.2.3 Encapsulando com Docker

Após finalizar e verificar a automação dos cenários de teste com o Cypress, iniciou-se a criação de uma imagem Docker para o projeto. A imagem Docker permite encapsular o ambiente completo do Cypress, tornando-o portátil e fácil de executar em outras máquinas. Para isso, foi criado um arquivo Dockerfile no projeto, que especifica as instruções necessárias para construir a imagem Docker. O Dockerfile é um arquivo de configuração que define todas as etapas para empacotar o ambiente e suas dependências, garantindo que o projeto seja executado de maneira consistente e independente da máquina onde está rodando.

Com a imagem Docker criada, foi possível executar o projeto de testes automatizados do Cypress diretamente através dela, permitindo a execução simplificada e encapsulada dos testes. Essa abordagem facilita a implementação de uma pipeline contínua, possibilitando que os testes sejam automaticamente executados em qualquer ambiente onde a imagem Docker esteja disponível, promovendo praticidade e reprodutibilidade no desenvolvimento e testes do aplicativo.

Após a execução bem-sucedida da imagem Docker localmente, foi necessário publicá-la na nuvem, utilizando o Docker Hub. O Docker Hub é uma plataforma de hospedagem que permite armazenar e distribuir imagens Docker, tornando-as acessíveis para download em qualquer ambiente de forma centralizada e organizada. Ao disponibilizar a imagem no Docker Hub, obtém-se um endereço para download, o que elimina a necessidade de manter o projeto de testes localmente. No contexto deste projeto, essa abordagem oferece várias vantagens: permite que qualquer máquina com acesso ao Docker Hub possa executar os testes sem configurações adicionais, garante consistência e reprodutibilidade ao evitar a dependência de ambientes locais, e facilita a colaboração entre equipes, uma vez que a imagem já contém todas as dependências e configurações necessárias.

5.2.4 Jenkins e Pipelines

Para implementar a pipeline contínua, foi utilizada a ferramenta Jenkins. Primeiramente, foi realizada a instalação do Jenkins em um sistema Linux, configurando a porta 8081 para acesso local. Durante o processo de instalação, é solicitado que se crie um usuário, onde foi escolhido o "admin" para administração do ambiente. Jenkins instala automaticamente alguns plugins essenciais, mas para atender aos requisitos específicos deste projeto, foram instalados outros plugins adicionais: o AnsiColor, que facilita a leitura dos logs gerados pela execução das pipelines, corrigindo problemas de formatação, e o Git Plugin, que permite a integração direta com o repositório Git, essencial para o versionamento e atualização do projeto.

Após a configuração inicial, foi acessado o Jenkins em localhost:8081 com o usuário "admin". No ambiente do Jenkins, foi criada uma nova pipeline, chamada *appcancer*, projetada para automatizar o processo de teste. As pipelines do Jenkins são configuradas por meio de arquivos de script chamados Jenkinsfiles, que contêm o código necessário para definir as etapas do pipeline. O arquivo possui uma estrutura declarativa, dividida em estágios que representam cada etapa do pipeline. Em um Jenkinsfile típico, utiliza-se uma estrutura de código que começa com a definição de um

pipeline, seguido pelos blocos agent, stages, e steps. No contexto deste projeto, o Jenkinsfile é configurado para: baixar automaticamente a imagem Docker do Docker Hub, executar a imagem em um ambiente isolado, e salvar as evidências de cada cenário de teste executado, podendo ser visto na Figura 3. Essa estrutura permite uma automação completa e confiável dos testes, garantindo que cada etapa do pipeline seja realizada com precisão e que os resultados estejam disponíveis para análise, promovendo a integração contínua e a eficiência no fluxo de trabalho de testes.

```
pipeline {
    agent any
    stages {
        stage('Baixar Imagem Docker') {
            steps {
                sh 'docker pull lucasrdmelo/cypress-tests:1.0'
            }
        }
        stage('Executar Container Docker') {
            steps {
                ansiColor('xterm') {
                    sh '''
                        docker run --rm --add-host=host.docker.internal:host-gateway \
                        -v $WORKSPACE/videos:/e2e/cypress/videos \
                        lucasrdmelo/cypress-tests
                    '''
                }
            }
        }
        stage('Arquivar Vídeos') {
            steps {
                archiveArtifacts artifacts: 'videos/**/*.mp4', allowEmptyArchive: true
            }
        }
    }
}
```

Figura 3. JenkinsFile

Fonte: Elaborada pelo autor

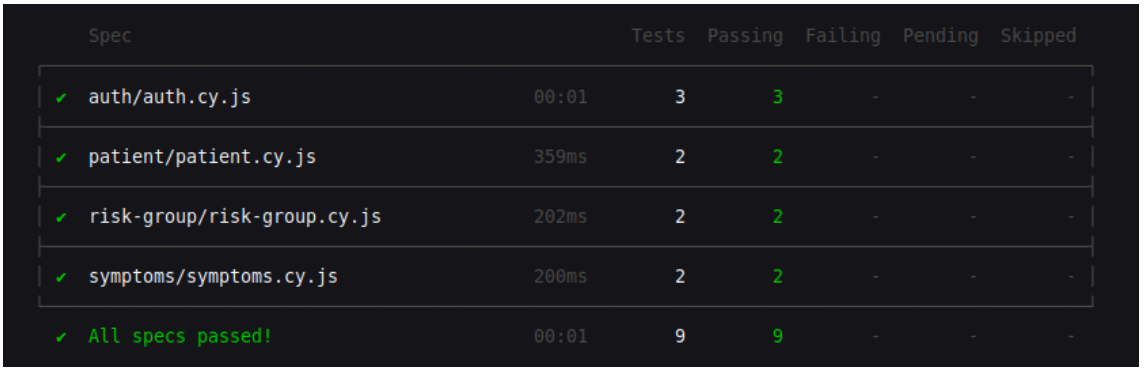
Dentro da configuração da pipeline no Jenkins, foi possível integrar o projeto diretamente com o repositório no GitLab, garantindo que o Jenkins tenha sempre acesso à versão mais recente do Jenkinsfile e do código de testes. Para isso, foi passado o endereço do repositório GitLab, juntamente com o caminho específico do Jenkinsfile dentro do projeto. Dessa forma, a cada execução da pipeline, o Jenkins utiliza o código mais atualizado disponível no repositório, promovendo uma automação contínua e confiável. Para possibilitar esse acesso, foi necessário armazenar as credenciais de acesso ao GitLab dentro do Jenkins.

5.2.5 Execução

Com o projeto de testes desenvolvido, a imagem Docker criada e a pipeline configurada para baixar e executar essa imagem, resta apenas iniciar a execução da pipeline no Jenkins. Ao executar a pipeline, o Jenkins gera logs detalhados de cada processo realizado, o que permite acompanhar todas as etapas da execução diretamente na interface de monitoramento do Jenkins.

Uma das vantagens do Cypress é que ele pode ser executado sem abrir sua interface gráfica, gerando um relatório direto nos logs do Jenkins. Esse relatório que

pode ser visto na figura 4, inclui informações fundamentais, como os resultados de cada teste, o tempo necessário para a integração com a API, e um resumo dos cenários de teste executados. Além disso, o Cypress possui a capacidade de gerar evidências em vídeo durante a execução dos cenários de teste, capturando em tempo real o processo de validação de cada endpoint. A cada execução da pipeline, todos esses registros são armazenados automaticamente, oferecendo um histórico completo das execuções.



Spec		Tests	Passing	Failing	Pending	Skipped
✔ auth/auth.cy.js	00:01	3	3	-	-	-
✔ patient/patient.cy.js	359ms	2	2	-	-	-
✔ risk-group/risk-group.cy.js	202ms	2	2	-	-	-
✔ symptoms/symptoms.cy.js	200ms	2	2	-	-	-
✔ All specs passed!	00:01	9	9	-	-	-

Figura 4. Resumo dos resultados dos cenários de teste

Fonte: Elaborada pelo autor

É importante destacar que todo o desenvolvimento e configuração foram realizados localmente na máquina utilizada para o projeto. Idealmente, o Jenkins deveria estar hospedado em um ambiente dedicado e acessível pela equipe, o que simplificaria o processo de validação da aplicação após cada commit ou merge no repositório. Na prática, com o Jenkins configurado em um servidor próprio, qualquer desenvolvedor – ou mesmo membros de outras áreas – poderia acessar o Jenkins através de um endereço remoto, realizar o login e iniciar a execução da pipeline com apenas alguns cliques.

Essa abordagem elimina a necessidade de configurar o projeto de testes localmente, tornando o processo de execução dos testes de API acessível e prático para toda a equipe, independentemente do conhecimento técnico específico de cada usuário. Essa facilidade promove uma cultura de colaboração e integração contínua, onde os testes de validação podem ser realizados por qualquer pessoa envolvida no projeto, garantindo que a qualidade da aplicação seja mantida de forma ágil e acessível a todos.

A clareza e acessibilidade dos relatórios gerados ao final das execuções são fundamentais para assegurar que os resultados dos testes sejam compreensíveis e acionáveis por todas as partes interessadas. Um relatório bem estruturado não apenas fornece uma visão consolidada dos cenários testados e seus resultados, mas também contribui para uma comunicação eficaz entre os membros da equipe de desenvolvimento e qualidade, independentemente de seu nível de familiaridade com as ferramentas utilizadas. Além disso, relatórios claros e detalhados facilitam a identificação rápida de problemas, otimizando os esforços de correção e promovendo uma tomada de decisão mais informada e ágil

6. Validação e resultados

A validação deste projeto foi realizada com o apoio de um desenvolvedor envolvido na criação da API do aplicativo AppCancer. Como o Jenkins ainda não está hospedado em um ambiente remoto, foi necessário que o desenvolvedor instalasse o Jenkins localmente em sua máquina. Uma vez configurado, ele pôde acessar e executar a

pipeline previamente estruturada, obtendo com sucesso os resultados dos testes automatizados diretamente pelos logs e vídeos gerados pelo Cypress. Esse processo foi realizado sem que o desenvolvedor precisasse configurar o projeto de teste ou possuir um ambiente específico para a execução, já que toda a infraestrutura e dependências estavam encapsuladas na imagem Docker do projeto.

A experiência demonstrou que o desenvolvedor, ao realizar modificações na API, pode facilmente validar suas atualizações acessando o Jenkins, fazendo login e selecionando a pipeline de testes. Dessa forma, a execução dos testes é feita automaticamente, sem a necessidade de conhecimentos aprofundados sobre o Cypress ou sobre os detalhes técnicos do ambiente de testes.

Os resultados foram bastante satisfatórios, pois agora o desenvolvedor não precisa de conhecimento específico sobre a ferramenta de testes para garantir a qualidade da API. Isso reduz o tempo necessário para validar alterações, permitindo que o foco permaneça no desenvolvimento e aprimoramento da API. O sistema automatizado mostrou-se eficaz em atender à necessidade de execução simplificada e confiável de testes, aumentando a eficiência do processo de desenvolvimento e liberando a equipe para se concentrar em atividades de maior valor.

7. Considerações Finais

Este trabalho ressalta a eficácia e os benefícios da integração entre Cypress, Docker e Jenkins na automação de testes de API em um ambiente DevOps. O projeto atingiu seu objetivo ao desenvolver um sistema de testes encapsulado, possibilitando a execução contínua e automatizada dos testes através de uma pipeline. Ao encapsular o ambiente de testes com Docker, assegurou-se a portabilidade e consistência dos testes, o que simplifica sua replicação em diferentes máquinas e ambientes.

A utilização do Jenkins permitiu a criação de uma pipeline contínua, que automatiza as etapas de download, execução e validação dos testes, gerando relatórios e vídeos com as evidências dos resultados. Essa estrutura mostrou-se particularmente eficaz ao dispensar a configuração local do projeto de testes, economizando tempo e evitando a curva de aprendizado de novas ferramentas para os desenvolvedores.

Os resultados obtidos indicam que a solução proposta não só otimiza o fluxo de trabalho da equipe como também contribui para uma cultura de integração e entrega contínuas, alinhando-se às melhores práticas de desenvolvimento ágil. O projeto, embora realizado localmente, mostrou o potencial para ser expandido a um ambiente remoto e escalável, onde o Jenkins poderia ser acessado de forma centralizada por toda a equipe.

Para trabalhos futuros, uma melhoria significativa seria a hospedagem do Jenkins em um ambiente dedicado, permitindo que ele seja acessado e gerenciado de forma centralizada e segura. Além disso, uma expansão interessante para a pipeline do Jenkins seria incluir uma etapa de build e deploy do ambiente de homologação ou produção do aplicativo "AppCancer". Com essa configuração, sempre que a API fosse implantada ou atualizada, os testes seriam executados automaticamente, validando de imediato a nova versão e assegurando que esteja funcionando conforme esperado.

Outro aprimoramento futuro seria aumentar a cobertura dos cenários de testes automatizados para a API. Atualmente, o projeto conta com um número limitado de cenários, já que o foco principal não foi a validação exaustiva do aplicativo de prevenção de câncer, mas sim a criação de uma integração entre três tecnologias: Cypress, Docker e Jenkins. Essa integração visa facilitar a execução dos testes e a obtenção de resultados de forma mais eficiente. Ampliar o conjunto de testes permitiria uma análise mais detalhada e completa das funcionalidades da API, garantindo maior cobertura e segurança no processo de validação.

Referências

Mironov, O. *DevOps Pipeline with Docker*, 2018. Disponível em: https://www.theseus.fi/handle/10024/143357. Acesso em: 22 out. 2024.

Thekkan, J., & Anuar, S. *Modern Web Automation with Cypress.io*, 2022. Disponível em: https://oiji.utm.my/index.php/oiji/article/view/229. Acesso em: 22 out. 2024.

Ferreira, *Desenvolvimento, testes e qualidade de software*. 2010.

FasterCapital, *Alcançando integração perfeita com desenvolvimento ágil.* Disponível em: https://fastercapital.com/pt/contente/Alcancando-integracao-perfeita-com-desenvolvimento-agil.html. Acesso em: 5 mar. 2024.

Portes, P. D. *Docker – A Revolução da Infraestrutura de Sistemas da Tecnologia da Informação*. Anais do EVINCI-UniBrasil, 8(2), p. 82-82, 2022.

Boaglio, F. *Jenkins: Automatize tudo sem complicações*. Editora Casa do Código, 2016.

Benitti, *Objetos de Aprendizagem para Teste de Software*. Disponível em: https://www.inf.ufsc.br/~fabiane.benitti/byebug/. Acesso em: 6 nov. 2024.

Valente, M. T. *Engenharia de Software Moderna*. 2020. Disponível em: https://engsoftmoderna.info/. Acesso em: 6 nov. 2024.

Rolt, (2024). *O que é a Pirâmide de Testes?* . Disponível em: https://blog.onedaytesting.com.br/piramide-de-teses/ . Acesso em: 23 nov. 2024.