CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA
FACULDADE DE TECNOLOGIA DE SÃO ROQUE
CURSO SUPERIOR TECNOLÓGICO DE SISTEMAS PARA INTERNET

FELIPE DOMINGUES DE ALMEIDA

TESTES AUTOMATIZADOS UTILIZANDO CYPRESS FRAMEWORK

São Roque
2023

CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA
FACULDADE DE TECNOLOGIA DE SÃO ROQUE
CURSO SUPERIOR TECNOLÓGICO DE SISTEMAS PARA INTERNET

FELIPE DOMINGUES DE ALMEIDA

TESTES AUTOMATIZADOS UTILIZANDO CYPRESS FRAMEWORK

Relatório Técnico apresentado à Faculdade de Tecnologia São Roque, como parte dos requisitos necessários para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Prof. Esp. José Luis Caetano Ribeiro Junior

São Roque
2023

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Domingues de Almeida, Felipe
Testes automatizados utilizando Cypress framework

São Roque, 2023.

49f.

Trabalho de Graduação – Curso de Tecnologia em Sistema para Internet

FATEC de São Roque, 2023.

Orientador: Prof. Esp. José Luis Caetano Ribeiro Junior.

REFERÊNCIA BIBLIOGRÁFICA –

Domingues de Almeida, Felipe. **Automação de testes utilizando Cypress Framework** 2023.49f. Trabalho de Graduação - FATEC de São Roque.

CESSÃO DE DIREITOS

NOME DO AUTOR: Felipe Domingues de Almeida
TÍTULO DO TRABALHO: Automação de testes utilizando Cypress Framework
TIPO DO TRABALHO/ANO: Trabalho de Graduação / 2023.

_____

Felipe Domingues de Almeida
2650831723027

DEDICATÓRIA

Dedico esse trabalho principalmente aos meus pais, que em nenhum momento deixaram de acreditar ou me apoiar em minhas decisões. Mesmo nos momentos mais difíceis acreditaram em mim.

RESUMO

O relatório técnico aborda conceitos básicos sobre testes automatizados e explica a estrutura do framework Cypress, além de apresentar exemplos práticos de testes utilizando essa ferramenta. São mostrados os benefícios dos testes automatizados, como a redução de erros e custos de manutenção, além de fornecerem feedback imediato para os desenvolvedores. Este trabalho apresenta uma introdução aos testes automatizados utilizando o framework Cypress. Os testes automatizados são uma prática essencial para garantir a qualidade e a estabilidade de aplicações web. O Cypress é uma ferramenta popular para a escrita e execução de testes automatizados, sendo conhecido por sua facilidade de uso e rapidez na detecção de erros.

Palavras-chave: Testes automatizados, Cypress, Qualidade de Software, Estabilidade, Detecção de erros.

LISTA DE FIGURAS

Figura 1 - Visão geral da apresentação de resultados de exames na aplicação 21
Figura 2 - Exemplo de utilização do método describe. 23
Figura 3 - Exemplo de criação de um método command 24
Figura 4 - Exemplo de reutilização de um método command em um cenário de teste. 24
Figura 5- Elementos mapeados através de CSS Selector 26
Figura 6- Mapeando elementos através da ferramenta Selector Playground 27
Figura 7 - Estrutura da massa de dados no formato JSON 29
Figura 8 - Utilização da massa de dados em cenário de teste. 29
Figura 9 - Casos de testes separado em métodos it 30
Figura 10- Exemplo de teste de API utilizando Cypress 31
Figura 11 - Seletor de diferentes navegadores na interface gráfica do Cypress 32
Figura 12 - Estado da aplicação antes de digitar um texto no campo 33
Figura 13 - Estado da aplicação após digitar um texto no campo 33
Figura 14- Interação com a timeline com o passo a passo do teste 34
Figura 15- Exemplo de capturas de tela dos cenários de teste executados 35
Figura 16- Execução dos casos de testes da funcionalidade Login 36
Figura 17- Execução dos casos de testes da funcionalidade Listagem de Pacientes 36
Figura 18 - Arquivo JenkinsFile contendo todos os passos a serem executados no pipeline de testes 38
Figura 19 - Integração da esteira de testes com ferramenta de comunicação Slack 39
Figura 20 - Visão geral do relatório de testes automatizados com Allure Reports 40
Figura 21 - Visão detalhada dos testes automatizados no relatório 41
Figura 22 - Visão de gráficos do relatório de testes automatizados 42

LISTA DE SIGLAS E ABREVIATURAS

API	Application Programming Interface
BDD	Behavior Driven Development
C	Linguagem de programação C
CD	Continuous Delivery
CI	Continuous Integration
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ISTQB	International Software Testing Qualifications Board
JAVA	Linguagem de Programação Java
JIT	Just in Time
JSON	JavaScript Object Notation
PERL	Linguagem de programação Perl
TDD	Test Driven Development

LISTA DE TABELAS

Tabela 1 - Escopo dos testes a serem automatizados na aplicação.....22
Tabela 2 - Exemplos de command actions disponíveis no Cypress 25
Tabela 3 - Exemplos de assertions e para que são utilizados 28

SUMÁRIO

1. INTRODUÇÃO 7

1.1. Objetivos 7

2. TESTE DE SOFTWARE 8

2.1. O que é teste de software? 8

2.2.1. Testes Unitários 9

2.2.2 Testes de Integração 9

2.2.3 Testes de Aceitação 9

2.2.4 Testes de Performance 10

2.2.5 Testes de Regressão 10

2.3. Testes End-to-End 10

2.4. Testes de API 11

2.4.1 Vantagens dos testes de API 12

2.4.2 Desvantagens dos testes de API 12

2.4.3 Tipos de testes de API 12

2.5. BDD 13

2.6. Erro defeito e falha 14

2.7. Execução dos casos de teste 15

3. AUTOMAÇÃO DE TESTES 16

3.1. O que é automação de testes 16

3.2. Tipos de testes automatizados 17

3.3 Cypress Framework 18

3.4. JavaScript 19

3.5. Jenkins 19

4. PROTOTIPAÇÃO 21

4.1. Aplicação 21

4.2. Escopo dos testes automatizados 21

4.3. Desenvolvimento dos scripts de testes automatizados 22

4.3.1 Método describe 23

4.3.2 Commands 23

4.3.3 Interações com o navegador 24

4.3.4 Mapeando elementos 25

4.3.5 Assertions 27

4.3.6 Massa de dados 28

4.3.7 Método It 29

4.4. Testes de API utilizando Cypress 30

4.5. Execução dos scripts de testes automatizados 31

4.5.1 Execução em múltiplos navegadores 32

4.5.2 Cypress Time Travel 32

4.5.3 Timeline de execução dos testes 34

4.6. Evidências dos cenários de teste 34

4.7. Execução do escopo de testes 35

4.8 Integração dos testes em uma pipeline utilizando Jenkins 37

4.9. Relatório de execução dos testes 39

CONSIDERAÇÕES FINAIS 43

REFERÊNCIAS 44

1. INTRODUÇÃO

Com a crescente demanda de desenvolvimento de software, é cada vez mais necessário garantir que o produto final esteja livre de erros e atenda aos requisitos do cliente. Uma das formas mais eficazes de garantir isso é através dos testes automatizados, que permitem que os desenvolvedores testem suas aplicações de forma mais rápida e eficiente, garantindo que todas as funcionalidades estejam funcionando corretamente (CRISPIN; GREGORY, 2008).

Os testes automatizados são uma parte fundamental da cultura de desenvolvimento de software moderna (MOLINARI, 2010), e o Cypress é uma opção atraente para equipes de desenvolvimento que buscam uma ferramenta poderosa e fácil de usar. Com este trabalho, é possível compreender melhor a importância dos testes automatizados e como o Cypress pode ser utilizado para garantir a qualidade e estabilidade de aplicações web.

Neste contexto, o Cypress Framework tem se destacado como uma das principais ferramentas para a realização de testes automatizados em aplicações web. Com sua sintaxe fácil de usar, alta velocidade de execução e a possibilidade de visualizar o processo de testes em tempo real, o Cypress tem sido adotado por diversas empresas como uma ferramenta essencial para garantir a qualidade de seus produtos.

1.1. Objetivos

Este trabalho tem como objetivo apresentar uma visão geral sobre o Cypress Framework e como ele pode ser utilizado para realizar testes automatizados em aplicações web. Serão abordados desde conceitos básicos como a estrutura de pastas e arquivos do Cypress, até funcionalidades mais avançadas como a realização de testes em diferentes navegadores e integração com outras ferramentas de desenvolvimento.

Ao longo do trabalho, serão apresentados exemplos práticos de como utilizar o Cypress para testar diversas funcionalidades de uma aplicação web, como formulários, validações de campos, navegação entre páginas e interação com elementos da página. Também serão apresentados exemplos de como escrever testes automatizados que simulem a interação do usuário com a aplicação, garantindo que todas as funcionalidades estejam funcionando corretamente e sem erros.

Por fim, serão discutidos os principais benefícios e desafios da utilização do Cypress Framework para testes automatizados em aplicações web, destacando a importância de se investir em ferramentas de qualidade para garantir a satisfação do cliente e o sucesso do produto final.

2. TESTE DE SOFTWARE

2.1. O que é teste de software?

O teste de software é uma atividade essencial no processo de desenvolvimento de software. É uma maneira de avaliar a qualidade do software e garantir que ele atenda aos requisitos do usuário e esteja livre de defeitos. O teste de software é realizado em diferentes etapas do ciclo de vida do software e é feito por diferentes perfis de profissionais, desde analistas de teste até usuários finais (MOLINARI, 2008).

Como descrito por Myers et al. (2011), o objetivo do teste de software é encontrar falhas no software antes que ele seja lançado. Uma falha é uma discrepância entre o comportamento real e o esperado do software. O processo de teste pode ser manual ou automatizado. No teste manual, um testador executa o software e verifica se ele se comporta conforme esperado. No teste automatizado, um software é usado para executar testes repetitivos em um sistema.

Existem vários tipos de teste de software, como teste de unidade, teste de integração, teste de sistema e teste de aceitação. O teste de unidade é realizado nos componentes individuais do software, enquanto o teste de integração é realizado na integração desses componentes. O teste de sistema é realizado no sistema completo e o teste de aceitação é realizado pelos usuários finais para avaliar se o software atende aos requisitos.

O teste de software também pode ser categorizado em termos de estratégia de teste. Uma estratégia de teste é um plano geral para testar o software. Algumas estratégias de teste comuns incluem teste de caixa-preta, teste de caixa-branca e teste de aceitação do usuário. No teste de caixa-preta, o testador não tem conhecimento interno do software e se concentra no comportamento externo do sistema. No teste de caixa-branca, o testador tem acesso ao código-fonte do software e se concentra na lógica interna do sistema. No teste de aceitação do usuário, os usuários finais testam o software para garantir que ele atenda aos requisitos.

É importante notar que o teste de software não é uma garantia de qualidade. Ele pode identificar problemas no software, mas não pode garantir que não haja mais falhas. Além disso, o teste de software pode ser caro e demorado, por isso é importante equilibrar o tempo e os recursos disponíveis com o nível de teste necessário para o software.

Em resumo, o teste de software é uma atividade essencial no processo de desenvolvimento de software. É uma maneira de avaliar a qualidade do software e garantir que ele atenda aos requisitos do usuário minimize consideravelmente possíveis defeitos no software, assim diminuindo custos e riscos para quem o desenvolve. Existem vários tipos de teste de software, estratégias e abordagens, que podem ser usados para testar o software de maneira abrangente. O processo de teste de software deve ser integrado com o processo de

desenvolvimento de software e ser conduzido desde o início do ciclo de vida do software. Além disso, é importante que o processo de teste seja documentado adequadamente e que os resultados dos testes sejam avaliados para identificar áreas que precisam de mais testes ou correção. Por fim, é essencial que os testadores tenham habilidades e conhecimentos específicos para realizar o teste de software de maneira eficaz.

2.2. Tipos de testes de software

Os testes de software são uma das principais atividades realizadas durante o desenvolvimento de software, e são cruciais para garantir que o produto final atenda aos requisitos de qualidade e funcionalidade (MOLINARI, 2008). Existem diferentes tipos de testes que podem ser aplicados a um software, cada um com um objetivo específico e com diferentes níveis de complexidade. Neste artigo, serão apresentados os principais tipos de testes de software.

2.2.1. Testes Unitários

Os testes unitários são realizados em pequenas unidades de código, como funções e métodos. O objetivo desses testes é verificar se cada unidade de código está funcionando corretamente e produzindo o resultado esperado. Eles são realizados pelos desenvolvedores e devem ser escritos juntamente com o código da unidade a ser testada. Os testes unitários são importantes porque permitem detectar e corrigir erros de forma precoce, o que pode reduzir custos e aumentar a eficiência do desenvolvimento (FOWLER, 2014).

2.2.2 Testes de Integração

Os testes de integração são realizados para verificar se as diferentes unidades de código funcionam corretamente juntas. Eles são realizados após os testes unitários e podem ser realizados em diferentes níveis de integração, desde a integração entre duas unidades até a integração de todo o sistema. O objetivo desses testes é garantir que as diferentes partes do sistema se comuniquem e funcionem de forma integrada (PRESSMAN, 2016).

2.2.3 Testes de Aceitação

Os testes de aceitação são realizados para verificar se o software atende aos requisitos do cliente. Eles são realizados após os testes de integração e são realizados pelos usuários finais ou representantes do cliente. O objetivo desses testes é validar se o software atende às necessidades do cliente e se está pronto para ser lançado (SOMMERVILLE, 2011).

2.2.4 Testes de Performance

Os testes de desempenho são realizados para verificar se o software atende aos requisitos de desempenho, como tempo de resposta e capacidade de processamento. Eles são realizados utilizando ferramentas especializadas e simulam situações de uso do software sob carga pesada. O objetivo desses testes é garantir que o software possa lidar com o volume de uso previsto sem falhas ou degradação do desempenho (PRESSMAN, 2016).

2.2.5 Testes de Regressão

Os testes de regressão são realizados a cada lançamento de uma nova versão do software, tendo como objetivo testar novamente as funcionalidades já existentes no software, garantindo assim que as alterações provenientes da versão não afetaram a integridade do restante do software, e garantindo assim que o mesmo não regrediu em sua qualidade e confiabilidade (MOLINARI, 2008).

Em suma, existem diversos tipos de testes de software que podem ser aplicados durante o desenvolvimento de um sistema. Cada tipo de teste tem um objetivo específico e contribui para garantir a qualidade e a funcionalidade do software. Portanto, é importante que as equipes de desenvolvimento de software estejam cientes dos diferentes tipos de testes disponíveis e os apliquem adequadamente.

2.3. Testes End-to-End

Os Testes End-to-End, também chamado de Testes de Sistema, são uma técnica de teste que tem como objetivo avaliar o comportamento de um sistema como um todo, em um ambiente que se assemelhe ao máximo com o de produção. Essa técnica de teste avalia a integração entre as diferentes camadas do sistema, desde a camada de apresentação até a camada de armazenamento de dados. Os testes end-to-end podem ser utilizados em diferentes tipos de sistemas, como sistemas web, sistemas móveis, sistemas desktop, entre outros (RIOS; MOREIRA, 2013).

De acordo com o Syllabus da International Software Testing Qualifications Board (ISTQB, 2018), os testes de sistema são uma técnica de teste funcional que avalia o comportamento do sistema como um todo. Essa técnica é utilizada para garantir que o sistema atenda aos requisitos funcionais e não funcionais definidos para o projeto. Além disso, essa

técnica de teste também é utilizada para avaliar a usabilidade e a experiência do usuário do sistema.

Os testes end-to-end podem ser realizados de diferentes formas. Uma das formas mais comuns é a utilização de testes manuais, nos quais um testador realiza uma sequência de ações no sistema e avalia o comportamento do sistema em resposta a essas ações. Outra forma de realizar testes end-to-end é a utilização de testes automatizados, nos quais um conjunto de scripts é desenvolvido para simular a interação do usuário com o sistema e avaliar o comportamento do sistema em resposta a essas interações.

Apesar de ser uma técnica de teste importante, os testes end-to-end também apresentam algumas limitações. De acordo com Cohn (2009), os testes end-to-end podem ser custosos em termos de tempo e recursos, uma vez que exigem que o sistema seja executado em seu ambiente de produção e que os testadores realizem fluxos complexos de ações no sistema. Além disso, os testes end-to-end também podem ser limitados em sua cobertura, uma vez que nem todos os cenários de uso do sistema podem ser simulados, e quando feitos em scripts automatizados, estão sujeitos a erro no desenvolvimento do script, uma vez que o teste só reproduzirá o que está descrito na automação.

Em resumo, os testes end-to-end são uma técnica de teste importante para garantir a qualidade do software. Essa técnica de teste permite avaliar o comportamento do sistema em seu ambiente de produção e garantir que o sistema atenda aos requisitos funcionais e não funcionais definidos para o projeto. Apesar de apresentar algumas limitações, os testes end-to-end são essenciais para garantir a qualidade e a confiabilidade do sistema em produção.

2.4. Testes de API

Os testes de API são uma etapa crucial no desenvolvimento de software (NOGUEIRA, 2020). Eles permitem que os desenvolvedores verifiquem se suas APIs estão funcionando corretamente e atendendo às expectativas do usuário. Este relatório técnico tem como objetivo apresentar os principais conceitos sobre testes de API, suas vantagens, desvantagens e tipos de testes que podem ser realizados.

Testes de API são um tipo de teste de software que verifica se as APIs de um sistema estão funcionando corretamente. APIs são interfaces de programação que permitem que diferentes sistemas se comuniquem e compartilhem dados entre si. Os testes de API geralmente envolvem o envio de solicitações para as APIs e a verificação das respostas retornadas. Eles são realizados para garantir que as APIs estejam funcionando corretamente e atendendo às expectativas do usuário.

2.4.1 Vantagens dos testes de API

Os testes de API oferecem diversas vantagens em relação a outros tipos de testes de software. Entre as principais vantagens, podemos citar:

Cobertura mais ampla: os testes de API podem testar uma grande variedade de cenários de uso, sem a necessidade de interface gráfica ou interação com o usuário.

Menor tempo de execução: os testes de API geralmente são mais rápidos do que os testes de interface do usuário, pois não envolvem a renderização de elementos gráficos.

Facilidade de automação: os testes de API são facilmente automatizados, o que permite a execução rápida e repetitiva de testes.

Maior estabilidade: como as APIs são interfaces padronizadas, os testes de API podem ser mais estáveis em comparação com outros tipos de testes, que podem ser afetados por alterações na interface do usuário.

2.4.2 Desvantagens dos testes de API

Apesar das vantagens dos testes de API, também há algumas desvantagens a serem consideradas. Algumas das principais desvantagens são:

Menor cobertura funcional: os testes de API não podem testar todas as funcionalidades de um sistema, especialmente aquelas que dependem da interface do usuário.

Dificuldade de detecção de problemas visuais: como os testes de API não envolvem a interface gráfica, é mais difícil detectar problemas visuais ou de usabilidade.

Dependência de documentação precisa: para realizar testes de API efetivos, é necessário ter uma documentação precisa e atualizada sobre as APIs do sistema.

2.4.3 Tipos de testes de API

Existem diversos tipos de testes que podem ser realizados em APIs, dependendo dos objetivos do teste e do nível de detalhamento necessário. Alguns dos principais tipos de testes de API são:

Testes de unidade: focados na verificação de uma única função ou método da API.

Testes de integração: focados na verificação da interação entre diferentes partes da API.

Testes de contrato: focados na verificação do contrato de interface da API, incluindo a validação dos parâmetros de entrada e saída.

Testes de carga: focados na verificação do desempenho da API sob diferentes cargas de solicitações.

Testes de segurança: focados na verificação da segurança da API, incluindo a descrição dos tipos de testes de API...

Testes de regressão: focados na verificação de possíveis regressões que possam ter ocorrido em uma nova versão da API.

Testes de mock: utilizados para simular o comportamento da API em um ambiente de teste controlado.

Testes de usabilidade: focados na verificação da usabilidade da API, incluindo a facilidade de uso e clareza da documentação.

Testes de end-to-end: focados na verificação da funcionalidade completa do sistema, incluindo a interação entre diferentes partes da API e outros sistemas.

Os testes de API são essenciais para garantir que as APIs de um sistema estejam funcionando corretamente e atendendo às expectativas do usuário. Eles oferecem diversas vantagens, como cobertura mais ampla, menor tempo de execução e facilidade de automação. No entanto, também têm algumas desvantagens, como menor cobertura funcional e dificuldade de detecção de problemas visuais. Existem diversos tipos de testes de API, cada um com seus objetivos e níveis de detalhamento específicos. É importante escolher o tipo de teste adequado para cada situação, a fim de obter resultados precisos e confiáveis.

2.5. BDD

BDD é uma abordagem de desenvolvimento de software que coloca o comportamento do software como ponto central do processo. Ela tem como objetivo principal criar um ambiente colaborativo e focado no entendimento das necessidades do usuário e na entrega de valor. Essa metodologia tem ganhado cada vez mais adeptos por conta de sua eficiência em melhorar a comunicação entre as equipes de desenvolvimento e entre as equipes e os stakeholders (CRISPIN; GREGORY, 2008).

O BDD é um método derivado do TDD, mas com um foco maior no comportamento do sistema e menos no código. Ele utiliza uma linguagem natural para descrever o comportamento do sistema e, a partir disso, criam-se testes automatizados que validam esse comportamento. Essa abordagem é útil porque ajuda a garantir que o software desenvolvido atenda às expectativas dos usuários.

O BDD é composto por três elementos principais: a especificação, o código e o teste. A especificação é a descrição do comportamento que o sistema deve ter, escrita em uma linguagem natural. O código é a implementação do comportamento descrito na especificação.

E o teste é a validação de que o código implementa corretamente o comportamento descrito na especificação.

Através da utilização do BDD, em combinação com a linguagem Gherkin, também é possível escrever casos de testes que serão utilizados como documentação para a equipe de desenvolvimento, e auxiliarão no processo de testes manuais e automatizados, descrevendo os cenários que deverão ser testados em cada funcionalidade. A linguagem Gherkin utiliza de palavras chaves para descrever as ações feitas nesses cenários, sendo elas:

- Dado: Utilizado para descrever uma pré-condição para o cenário de teste, uma ação necessária antes de executar as ações do cenário a ser testado, por exemplo: “Dado possuir um usuário com tipo de acesso médico”.
- Quando: Utilizado para descrever as ações efetuadas no cenário de teste, simulando as ações que seriam tomadas pelo usuário para reproduzir o comportamento esperado, por exemplo: “Acessar a tela de login da aplicação”; “Preencher o campo de usuário com o CPF do médico”.
- Então: Utilizado para descrever quais as validações necessárias para garantir que o cenário de teste está se comportando ou não com o esperado da aplicação, sendo um dos passos mais importantes pois os testes devem garantir que os requisitos estão sendo atendidos. Exemplo: “Então validar que o login é efetuado com sucesso”; “Então validar que é exibida uma mensagem de erro para o usuário”.
- E: A conjunção é utilizada para quando é necessária mais de uma interação, seja ela em qualquer das etapas anteriores, visto que o “E” assume a ação da palavra-chave utiliza anteriormente. Por exemplo: “E preencher o campo de senha do usuário com uma senha inválida”.

2.6. Erro defeito e falha

Ao executar testes, sejam eles manuais ou automatizados, é importante compreender os conceitos de erro, defeito e falha. De acordo com Bastos et al. (2007). erro é um comportamento não planejado que ocorre durante a execução do software. Já o defeito é o resultado de um erro que não foi detectado e corrigido. E a falha é o resultado de um defeito que foi acionado, ou seja, é a manifestação do problema para o usuário final.

Para melhor entender esses conceitos, pode-se utilizar um exemplo simples: suponha que o software deve calcular o resultado da soma de dois números. Caso, ao executar o software, o resultado for diferente do esperado, então ocorreu um erro. Se o desenvolvedor não detectar

e corrigir esse erro, ele se torna um defeito. E se o usuário final executar o software e perceber que o resultado está incorreto, então ocorreu uma falha.

O objetivo do teste de software é durante o ciclo de desenvolvimento de software, encontrar esses problemas e apontar para correção o mais rápido possível, visto que quanto mais cedo, menor o custo de uma correção (MUSTAFA, K; KHAN, R, 2007).

2.7. Execução dos casos de teste

A execução dos casos de teste é uma das etapas fundamentais da automação de testes de software. Essa etapa consiste na execução dos testes automatizados com o objetivo de avaliar o comportamento do sistema em relação aos requisitos especificados. A automação de testes tem ganhado cada vez mais importância no processo de desenvolvimento de software, uma vez que permite reduzir os custos e o tempo necessários para a realização dos testes, além de aumentar a qualidade do produto.

De acordo com Molinari (2010), a execução dos casos de teste em um ambiente de automação pode ser dividida em três fases: preparação, execução e análise dos resultados. Na fase de preparação, são definidos os cenários de testes, os dados necessários para a execução dos testes e as configurações do ambiente. Na fase de execução, os casos de teste são executados automaticamente, e os resultados são armazenados em um relatório de testes. Por fim, na fase de análise dos resultados, os resultados são avaliados para verificar se o comportamento do sistema está de acordo com o esperado.

Segundo Molinari (op. cit.), a execução dos casos de teste deve ser realizada de forma sistemática e repetitiva, de modo a garantir a qualidade do software em diferentes cenários e condições. Para isso, é importante que os casos de teste sejam bem elaborados e que cubram todas as funcionalidades e cenários relevantes do sistema. Além disso, é fundamental que a equipe de testes tenha conhecimento sobre as técnicas e ferramentas de automação de testes, de modo a garantir a efetividade dos testes.

2.8. Evidências de testes

A automação de testes é uma técnica de teste de software que utiliza ferramentas de software para controlar a execução de testes e comparar os resultados esperados com os resultados obtidos (BEIZER, 1995). A automação de testes é amplamente utilizada na indústria de software para melhorar a eficiência do teste e reduzir o tempo necessário para executar testes repetitivos. No entanto, a eficácia da automação de testes depende da qualidade dos testes automatizados e da capacidade de gerar evidências confiáveis dos testes executados.

A evidência de teste é uma documentação que fornece informações sobre os testes executados e os resultados obtidos. A evidência de teste é importante para permitir a reprodução dos testes, para documentar o progresso dos testes e para identificar problemas no software. Na automação de testes, a evidência de teste é gerada automaticamente pela ferramenta de automação de testes. No entanto, a qualidade da evidência de teste gerada pela ferramenta de automação de testes é fundamental para garantir a eficácia da documentação dos testes executados e sua análise e manutenção, caso necessária.

3. AUTOMAÇÃO DE TESTES

3.1. O que é automação de testes

A automação de testes é uma prática importante na área de desenvolvimento de software que envolve a utilização de ferramentas e técnicas automatizadas para verificar a qualidade e funcionalidade de um software durante o processo de desenvolvimento. Segundo Beizer (1995), a automação de testes tem como objetivo principal acelerar o processo de testes e aumentar a eficácia e eficiência da detecção de erros e problemas em um software.

Segundo Martin (2013), a automação de testes pode ser realizada através da criação de scripts e casos de testes automatizados que podem ser executados repetidamente para testar as funcionalidades de um software. Além disso, a automação de testes também pode ser utilizada para realizar testes de carga e desempenho em um software.

De acordo com Pressman (2016), a automação de testes é importante para reduzir o tempo e o custo de teste, além de aumentar a qualidade do software e reduzir o risco de falhas em produção. A automação de testes também pode ajudar a melhorar a colaboração entre equipes de desenvolvimento e testes e a reduzir a dependência de testadores manuais.

Entretanto, é importante lembrar que a automação de testes não pode substituir completamente os testes manuais, pois ainda é necessário o envolvimento de testadores humanos para avaliar aspectos que não podem ser avaliados automaticamente, como a usabilidade e experiência do usuário. Segundo Molinari (2010), a automação de testes deve ser utilizada como uma ferramenta complementar aos testes manuais, visando aumentar a eficiência e eficácia dos testes realizados.

Em resumo, a automação de testes é uma prática importante na área de desenvolvimento de software que tem como objetivo acelerar o processo de testes e aumentar a eficácia e eficiência da detecção de erros e problemas em um software. Entretanto, é importante lembrar

que a automação de testes deve ser utilizada como uma ferramenta complementar aos testes manuais, visando aumentar a eficiência e eficácia dos testes realizados.

3.2. Tipos de testes automatizados

Os testes automatizados são uma prática comum no desenvolvimento de software, pois ajudam a garantir a qualidade do produto final. Eles permitem que os desenvolvedores possam executar testes de forma rápida e eficiente, além de proporcionar uma maior confiabilidade nos resultados obtidos.

Existem vários tipos de testes automatizados que podem ser realizados, sendo que cada um tem uma finalidade específica. Neste relatório técnico serão apresentados quatro tipos de testes: testes unitários, testes de integração, testes de aceitação e testes de regressão.

Os testes unitários são usados para testar o funcionamento de cada unidade individual do código, como métodos e funções, de forma isolada. Eles são importantes para garantir que o código de cada unidade está funcionando corretamente, além de permitir uma identificação mais fácil de possíveis erros e falhas. Segundo Fowler (2014), os testes unitários são uma das principais práticas de testes em desenvolvimento de software.

Os testes de integração, por sua vez, são usados para testar a integração entre as diferentes partes do sistema, como módulos e componentes. Eles ajudam a identificar problemas de comunicação e integração entre as diferentes partes do sistema, e a garantir que todas as partes estão funcionando corretamente em conjunto. De acordo com Pressman (2016), os testes de integração são essenciais para garantir a qualidade do sistema como um todo.

Os testes de aceitação, também conhecidos como testes funcionais, são usados para testar se o software atende aos requisitos do cliente e se está funcionando de acordo com as especificações. Eles são importantes para garantir que o software está cumprindo seu propósito e atendendo às necessidades do usuário final. Conforme explicado por Sommerville (2011), os testes de aceitação são uma etapa crucial na validação do software.

Por fim, os testes de regressão são usados para testar se as alterações e melhorias feitas no software não afetaram a funcionalidade já existente. Eles ajudam a garantir que as mudanças feitas não causaram problemas em outras partes do sistema. Segundo Molinari (2010), os testes de regressão são importantes para garantir a estabilidade e confiabilidade do software.

Em suma, os testes automatizados são uma etapa essencial no desenvolvimento de software, e cada tipo de teste tem sua importância específica na garantia da qualidade do produto. A utilização de diferentes tipos de testes automatizados, em conjunto, proporciona uma maior segurança e confiabilidade no software desenvolvido.

3.3 Cypress Framework

O framework Cypress é uma ferramenta de teste automatizado que permite aos desenvolvedores escreverem, executarem e depurarem testes de forma fácil e eficiente. Ele é amplamente utilizado para testar aplicações da web em ambientes de desenvolvimento ágeis. O Cypress oferece uma experiência de teste contínua e rápida, permitindo que os desenvolvedores realizem testes enquanto desenvolvem o código, o que ajuda a identificar e resolver problemas mais rapidamente.

O Cypress é um framework de teste de código aberto e gratuito que foi desenvolvido por Brian Mann em 2014. Desde então, ele tem sido utilizado por milhares de desenvolvedores em todo o mundo e se tornou um dos frameworks de teste mais populares para aplicações web (CYPRESS, 2022). O Cypress é construído com JavaScript e roda no navegador, permitindo que os desenvolvedores criem testes de forma fácil e flexível. Ele oferece uma variedade de recursos para simplificar a escrita de testes, incluindo seletores personalizados, comandos de teste personalizados, uma ferramenta de depuração integrada e uma interface de linha de comando.

O Cypress também possui uma arquitetura robusta que o torna ideal para testes de integração e ponta a ponta. Ele permite que os desenvolvedores escrevam testes que interagem com a aplicação da mesma maneira que um usuário final, testando a interface do usuário, interações do usuário e integrações de API. Isso ajuda a garantir que a aplicação funcione corretamente em todos os ambientes de produção e que a experiência do usuário seja consistente em todos os dispositivos.

Outra vantagem do Cypress é sua documentação abrangente e extensa. O site oficial do Cypress possui uma ampla variedade de tutoriais, documentação e exemplos de código que podem ajudar os desenvolvedores a começar rapidamente com o framework. Além disso, a comunidade de usuários do Cypress é ativa e engajada, oferecendo suporte e orientação para os desenvolvedores que precisam de ajuda.

O framework Cypress é uma ferramenta de teste automatizado altamente eficiente e flexível que oferece uma experiência de teste contínua e rápida para desenvolvedores. Sua arquitetura robusta e documentação extensiva o tornam uma opção popular para testes de integração e ponta a ponta em ambientes de desenvolvimento ágeis. Com sua comunidade ativa e engajada, o Cypress é uma opção atraente para desenvolvedores que desejam criar testes de forma fácil e eficiente para suas aplicações web.

3.4. JavaScript

JavaScript é uma linguagem de programação de alto nível que foi criada em 1995 por Brendan Eich na Netscape Communications Corporation. Desde então, ela se tornou uma das linguagens mais populares do mundo e é usada tanto para desenvolvimento front-end quanto back-end. É uma linguagem interpretada, o que significa que o código é executado linha a linha, sem a necessidade de compilação prévia.

A sintaxe do JavaScript é baseada em outras linguagens de programação, incluindo Java, C e Perl. Ela suporta tipos de dados como números, strings, arrays, objetos e booleanos, e tem recursos avançados, como funções de primeira classe, funções anônimas e fechamentos.

Uma das principais características do JavaScript é sua capacidade de interagir com elementos HTML e CSS. Isso permite que os desenvolvedores criem animações, atualizem conteúdo dinamicamente e adicionem interatividade aos seus sites. Além disso, a linguagem também é usada para criar aplicativos web, jogos e extensões de navegador.

No entanto, o JavaScript também tem suas desvantagens. Uma delas é que ele pode ser vulnerável a ataques de segurança, como ataques de injeção de código. Para lidar com isso, os desenvolvedores devem seguir as melhores práticas de segurança e usar bibliotecas e estruturas seguras.

Outra desvantagem é que, por ser interpretada, a execução de código JavaScript pode ser mais lenta do que a de linguagens compiladas. No entanto, as melhorias na tecnologia de JavaScript, como o JIT Compiler, ajudaram a melhorar a velocidade de execução.

O JavaScript é uma linguagem dinâmica e versátil que é amplamente utilizada no desenvolvimento web. É uma habilidade importante para desenvolvedores de front-end e back-end, e é provável que continue sendo uma linguagem popular no futuro (FLANAGAN, 2020).

3.5. Jenkins

Jenkins é uma ferramenta de integração contínua e entrega contínua (CI/CD) de código aberto que permite a automação de tarefas repetitivas em um processo de desenvolvimento de software. Essa ferramenta permite que os desenvolvedores possam integrar o código fonte de seus projetos em um repositório central, testar o código automaticamente, realizar o build do projeto e distribuir as aplicações de forma automatizada. O uso de Jenkins traz diversos benefícios para os projetos de desenvolvimento de software, tais como: aumento da produtividade, melhoria da qualidade do código, redução de erros, entre outros (DURVAL; MATOUSEK; HUMBLE, 2007).

Para entender melhor como funciona o Jenkins é importante conhecer alguns conceitos chave, tais como: integração contínua, entrega contínua e automação. Integração contínua (CI) é um processo em que os desenvolvedores enviam suas alterações de código para um repositório compartilhado várias vezes ao dia, o que permite a detecção de erros mais cedo no ciclo de desenvolvimento. A entrega contínua (CD) é a prática de entregar o código fonte de forma automatizada, rápida e segura para o ambiente de produção.

A configuração do Jenkins é realizada por meio de um arquivo de configuração (Jenkinsfile), onde é possível definir os passos para a execução de tarefas específicas, como a compilação, testes, deploy, entre outros. Além disso, o Jenkins pode ser integrado a outras ferramentas de gerenciamento de projetos e comunicação.

A utilização do Jenkins traz diversas vantagens para os desenvolvedores. Segundo Duvall et al. (2007), a integração contínua permite a detecção precoce de erros, o que reduz o custo de correção e melhora a qualidade do software. Além disso, a automação do processo de build e deploy reduz o tempo necessário para entregar novas funcionalidades para o usuário final. O uso do Jenkins também permite a criação de um ambiente de desenvolvimento mais colaborativo, em que os desenvolvedores podem compartilhar conhecimentos e experiências para aprimorar o processo de desenvolvimento.

Porém, é importante destacar que o uso do Jenkins requer uma boa estratégia de testes automatizados para garantir a qualidade do software entregue. Como aponta Boaglio (2016), é necessário definir um conjunto de testes unitários, testes de integração e testes funcionais para garantir que a aplicação esteja funcionando corretamente em todas as etapas do desenvolvimento.

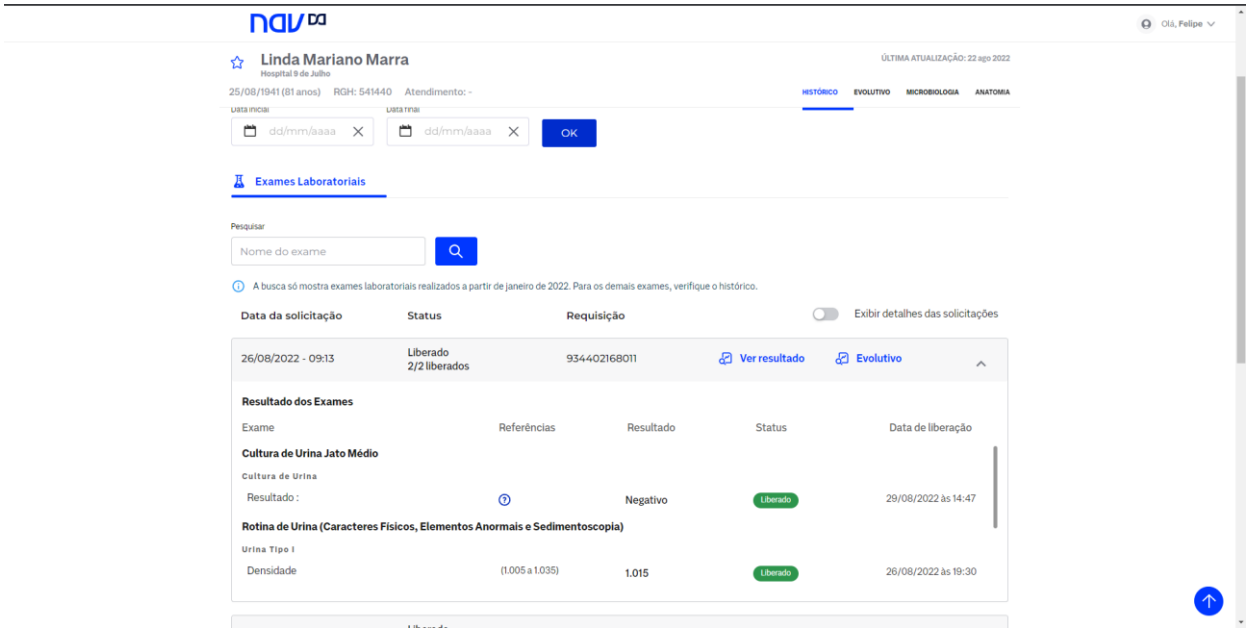
Em resumo, o Jenkins é uma ferramenta essencial para os projetos de desenvolvimento de software que buscam aumentar a produtividade, melhorar a qualidade do código e reduzir o tempo de entrega de novas funcionalidades. Seu uso requer uma boa estratégia de testes automatizados para garantir a qualidade do software entregue.

4. PROTOTIPAÇÃO

4.1. Aplicação

Para apresentação deste trabalho, os testes automatizados serão desenvolvidos e executados em uma aplicação real, desenvolvida por uma grande empresa de saúde em atividade no país. A aplicação apresenta uma solução para profissionais da saúde poderem visualizar resultados de exames de todos os pacientes com atendimentos em hospitais da marca ou que possuem contrato para utilização da aplicação. A Figura 1 apresenta uma visão geral de uma das funcionalidades da aplicação, onde é possível verificar os resultados de exames para um determinado atendimento de um paciente.

Figura 1 - Visão geral da apresentação de resultados de exames na aplicação



Fonte: Elaboração própria.

4.2. Escopo dos testes automatizados

Neste trabalho, para apresentação da ferramenta foram definidos escopos de testes de acordo com algumas das funcionalidades da aplicação, no caso em questão as funcionalidades Login e Listagem de Pacientes, que de forma geral podem apresentar todo o contexto de desenvolvimento e execução dos testes automatizados utilizando Cypress, de forma com que os testes garantam o padrão esperado para os cenários listados na Tabela 1.

Tabela 1 - Escopo dos testes a serem automatizados na aplicação

Funcionalidade	Cenário	Resultado esperado para o cenário
Login	Login com usuário Admin	Validar login efetuado com sucesso para usuário com tipo de acesso ‘Admin’
Login	Login com usuário Médico	Validar login efetuado com sucesso para usuário com tipo de acesso ‘Médico’
Login	Mensagem de erro ao não preencher CPF no Login	Validar exibição de mensagem de mensagem de erro ao tentar se logar sem preencher CPF
Login	Mensagem de erro ao não preencher senha no Login	Validar exibição de mensagem de erro ao tentar se logar sem preencher senha
Listagem de pacientes	Carregamento da lista de pacientes	Validar carregamento com sucesso da lista de pacientes ao acessar um hospital
Listagem de pacientes	Busca de paciente por nome completo	Validar busca de paciente com sucesso utilizando nome completo do paciente
Listagem de pacientes	Busca de paciente por número de atendimento	Validar busca do paciente com sucesso utilizando o número de atendimento do paciente

Fonte**:** Elaboração própria.

4.3. Desenvolvimento dos scripts de testes automatizados

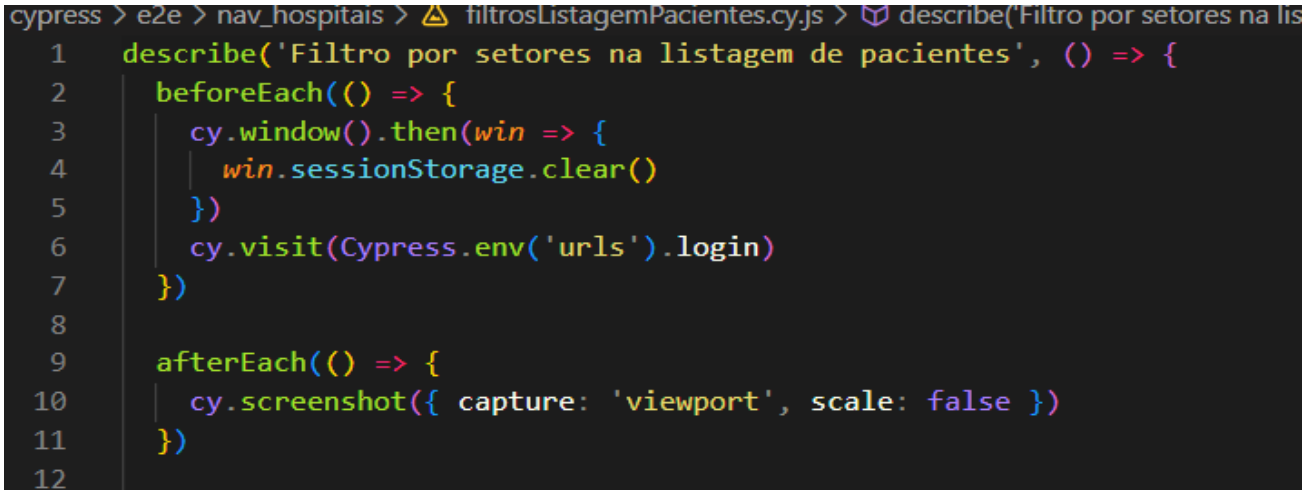
Para elaborar os testes automatizados planejados no escopo da Tabela 1, será utilizado o framework Cypress em conjunto com a linguagem de programação Javascript. Os testes são escritos de forma com que cada cenário de teste deva atendar as entradas e resultados esperados para o cenário em questão. Por exemplo, para o cenário “Busca de paciente por número de

atendimento", o cenário de teste deverá efetuar todo o fluxo mapeado para o cenário de teste, e garantir que a aplicação se comporta como o esperado.

4.3.1 Método describe

O desenvolvimento de scripts de testes em Cypress devem necessariamente começar com a utilização do método *describe*, que dentro do contexto do Cypress funciona como um agrupador para todos os testes de determinada funcionalidade. Todos os cenários de testes referentes a funcionalidade serão agrupados dentro desse método, possibilitando que ações necessárias antes ou depois dos casos de testes, como por exemplo acessar a página onde a funcionalidade será testada, sejam efetuadas para todos os testes dentro do método através dos comandos *beforeEach* e *afterEach*. Exemplo de utilização do método *describe* apresentado na Figura 2:

Figura 2 - Exemplo de utilização do método *describe*.

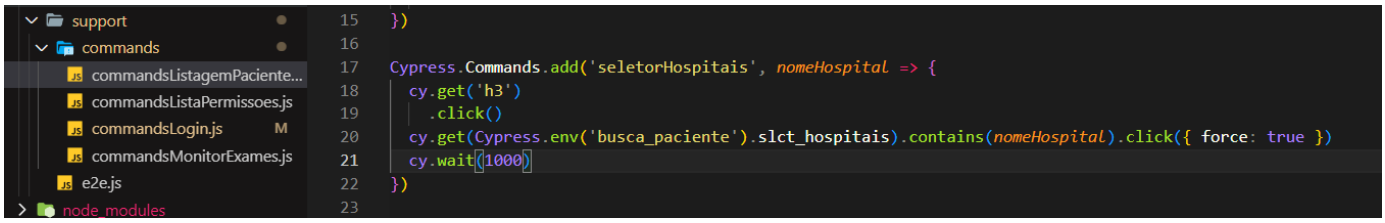


Fonte: Elaboração própria.

4.3.2 Commands

O Cypress nativamente possui uma funcionalidade chamada Commands, onde através de sua utilização é possível criar e encapsular métodos para ações que são utilizadas diversas vezes nos testes, como por exemplo uma ação de login na aplicação, que deverá ser reutilizada em todos os cenários que necessitem acessar a área logada. Dentro da pasta *commands* é possível criar arquivos para cada funcionalidade, e dentro deles criar métodos através do método *Cypress.Commands.add* e reutilizar esses métodos em qualquer um dos testes, conforme demonstrado nas figuras 3 e 4.

Figura 3 - Exemplo de criação de um método command



Fonte: Elaboração própria.

Figura 4 - Exemplo de reutilização de um método command em um cenário de teste.

```
it.only('Validar filtro por setor UTI Adulto - Hospital Santa Paula', () => {
  cy.preencherCamposLogin(Cypress.env('cpf_admin'), Cypress.env('loginUiPassword'))
  cy.selecionarHospitalLogin(Cypress.env('hospital_9_julho'))
  cy.seletorHospitais(Cypress.env('hospital_santa_paula'))
  cy.selecionarSetor()
  cy.get(Cypress.env('filtro_setor').setor_uti_adulto_santa_paula)
    .click()
  cy.get(Cypress.env('filtro_setor').btn_filtrar_setor)
    .click({ force: true })
  cy.validarTamanhoLista('20')
  cy.validarResultadoSetor(Cypress.env('setor_uti_santa_paula'))
})
```

Fonte: Elaboração própria.

4.3.3 Interações com o navegador

Os testes automatizados *End-to-End* tem como uma de suas premissas simular as interações de um usuário com a aplicação em um determinado fluxo, e assim, através de interações e validações, garantir que o comportamento está de acordo com o esperado. O Cypress de forma nativa possui diversos métodos que simulam essas interações com o navegador, chamados de *command actions*, onde a partir delas é possível reproduzir todo o fluxo esperado de um usuário da aplicação. Na tabela 2 serão listados alguns desses métodos e quais interações com o navegador são possíveis efetuar.

Tabela 2 - Exemplos de command actions disponíveis no Cypress

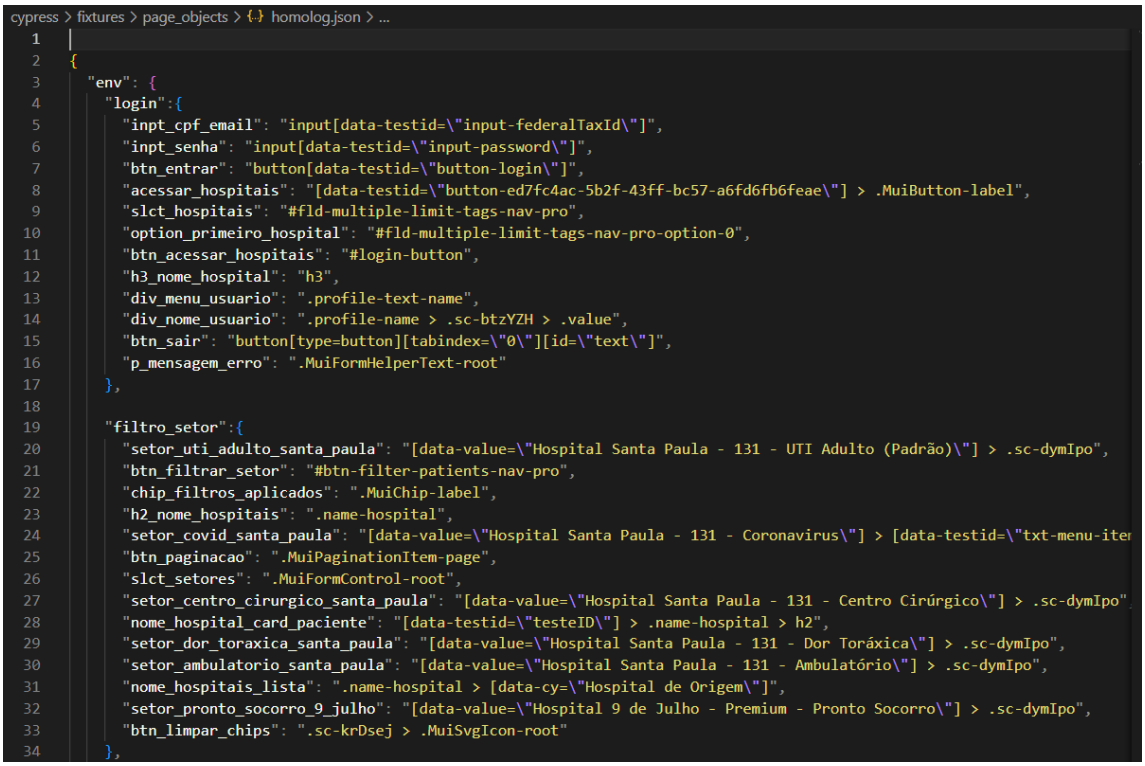
Command action	Interação efetuada com o navegador
.click()	Efetua o click do mouse em algum dos elementos da tela, seja um botão, link, div etc.
.rightclick()	Simula o click com o botão direito do mouse, caso seja necessário esse tipo de interação.
.type()	Simula a digitação em um campo de texto, podendo ser usado em formulários, campos do tipo input etc.
.clear()	Limpa o que foi digitado no campo de texto, podendo assim simular a limpeza de campos de formulários.
.check()	Marca como checados elementos do tipo checkbox.
.uncheck()	Remove a marcação de checado de um elemento do tipo checkbox.
.select()	Permite selecionar uma das opções e interagir com elementos do tipo dropdown.

Fonte: Elaboração própria.

4.3.4 Mapeando elementos

Para poder interagir com os elementos da tela utilizando a automação, é necessário mapear os elementos para possibilitar essa interação. O mapeamento de elementos é feito através dos através das tags e atributos de HTML e CSS utilizados para criar os elementos, por exemplo, para mapear um botão posso utilizar da tag button e uma das propriedades que identifiquem o botão com o qual quero interagir. As propriedades podem ser tanto o id, classe, nome etc. Esse tipo de mapeamento é conhecido como CSS Selector. Na figura 5 são demonstrados exemplos de elementos mapeados na automação através de CSS Selector. Utilizando Cypress também é possível encontrar elementos pelo texto do mesmo, com o comando *cy.contains*.

Figura 5- Elementos mapeados através de CSS Selector



cypress > fixtures > page_objects > {..} homolog.json > ...

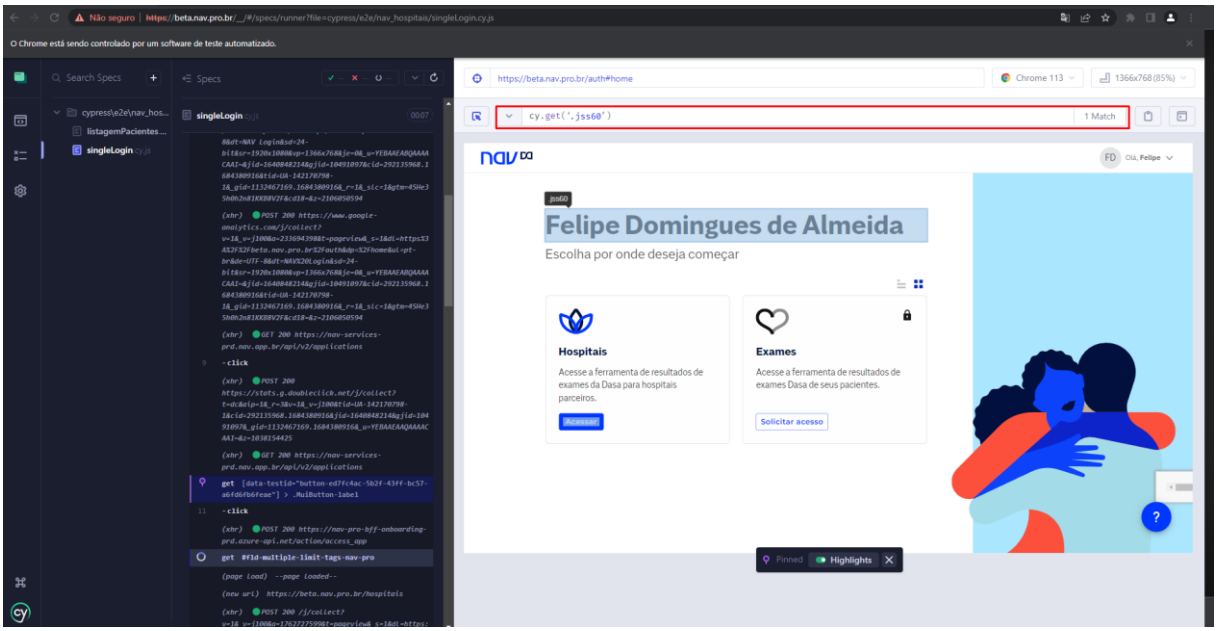
```
{
  "env": {
    "login":{
      "inpt_cpf_email": "input[data-testid=\"input-federalTaxId\"]",
      "inpt_senha": "input[data-testid=\"input-password\"]",
      "btn_entrar": "button[data-testid=\"button-login\"]",
      "acessar_hospitais": "[data-testid=\"button-ed7fc4ac-5b2f-43ff-bc57-a6fd6fb6feae\"] > .MuiButton-label",
      "slct_hospitais": "#fld-multiple-limit-tags-nav-pro",
      "option_primeiro_hospital": "#fld-multiple-limit-tags-nav-pro-option-0",
      "btn_acessar_hospitais": "#login-button",
      "h3_nome_hospital": "h3",
      "div_menu_usuario": ".profile-text-name",
      "div_nome_usuario": ".profile-name > .sc-btzYZH > .value",
      "btn_sair": "button[type=button][tabindex=\"0\"][id=\"text\"]",
      "p_mensagem_erro": ".MuiFormHelperText-root"
    },

    "filtro_setor":{
      "setor_uti_adulto_santa_paula": "[data-value=\"Hospital Santa Paula - 131 - UTI Adulto (Padrão)\"] > .sc-dymIpo",
      "btn_filtrar_setor": "#btn-filter-patients-nav-pro",
      "chip_filtros_aplicados": ".MuiChip-label",
      "h2_nome_hospitais": ".name-hospital",
      "setor_covid_santa_paula": "[data-value=\"Hospital Santa Paula - 131 - Coronavirus\"] > [data-testid=\"txt-menu-iter
      "btn_paginacao": ".MuiPaginationItem-page",
      "slct_setores": ".MuiFormControl-root",
      "setor_centro_cirurgico_santa_paula": "[data-value=\"Hospital Santa Paula - 131 - Centro Cirúrgico\"] > .sc-dymIpo"
      "nome_hospital_card_paciente": "[data-testid=\"testeID\"] > .name-hospital > h2",
      "setor_dor_toraxica_santa_paula": "[data-value=\"Hospital Santa Paula - 131 - Dor Toráxica\"] > .sc-dymIpo",
      "setor_ambulatorio_santa_paula": "[data-value=\"Hospital Santa Paula - 131 - Ambulatório\"] > .sc-dymIpo",
      "nome_hospitais_lista": ".name-hospital > [data-cy=\"Hospital de Origem\"]",
      "setor_pronto_socorro_9_julho": "[data-value=\"Hospital 9 de Julho - Premium - Pronto Socorro\"] > .sc-dymIpo",
      "btn_limpar_chips": ".sc-krDsej > .MuiSvgIcon-root"
    },
```

Fonte: Elaboração própria.

Uma vantagem de mapear elementos utilizando o Cypress, é que o mesmo disponibiliza uma ferramenta chamada "*Selector Playground*", onde é possível através da própria interface de execução, clicar em um elemento exibido na tela, e o próprio Cypress disponibilizará o seletor referente ao elemento selecionado, como demonstrado na figura 6, o que torna o processo de mapear elementos muito mais ágil e precisa.

Figura 6- Mapeando elementos através da ferramenta Selector Playground



Fonte: Elaboração própria.

4.3.5 Assertions

De acordo com o conceito de testes, todos os cenários devem ter uma ou mais validações, de forma com que seja validado que o resultado do cenário de teste está ou não de acordo com o esperado para o fluxo efetuado. O frakework Cypress utiliza a biblioteca *Chai*, uma das mais utilizadas da linguagem JavaScript, para efetuar essas validações, também chamadas de *assertions*. Através dessa biblioteca é possível efetuar diferentes tipos de comparações para garantir que o resultado do teste está de acordo com o esperado. Algumas das validações disponíveis são por exemplo validar que um elemento existe ou não na tela, que um texto está de acordo com o esperado, se um campo está vazio ou não, entre diversas outras. Na tabela 3 serão listadas algumas das validações mais utilizadas.

Tabela 3 - Exemplos de assertions e para que são utilizados

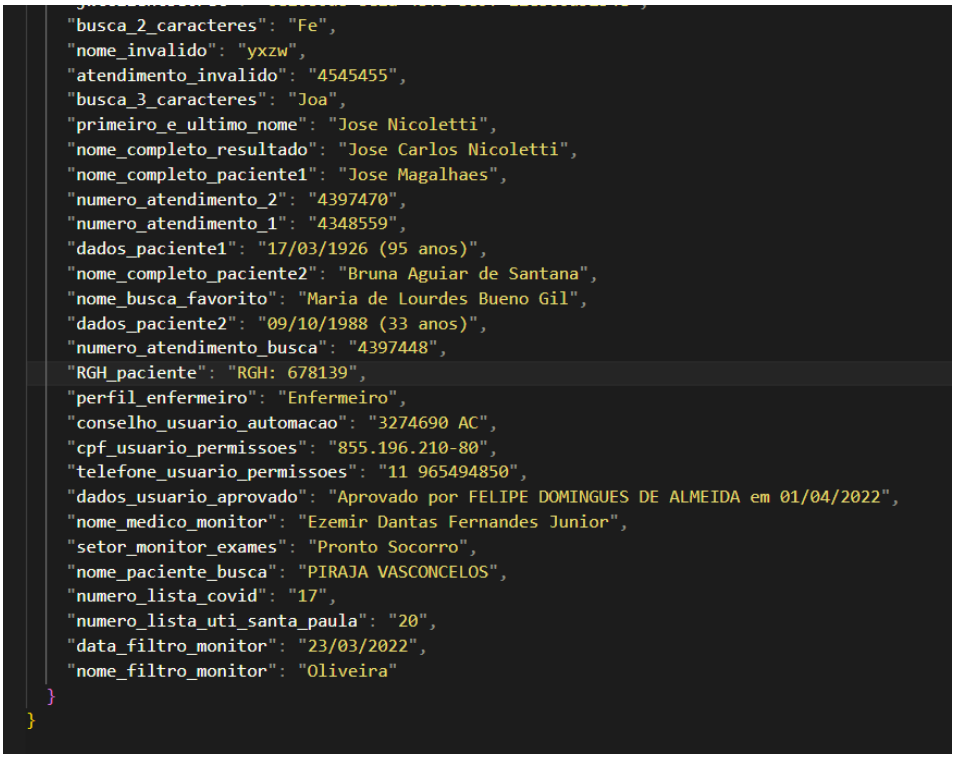
Assertion	Exemplo de uso	Para que é utilizada
Include	“Text field”. should(‘include’, ‘Text)	Validar que um elemento contenha determinado texto ou valor
Eql	1.should(‘eql’, 1)	Validar que um elemento tenha exatamente o mesmo valor e tipo da comparação.
Not	‘João’.should(‘not.equal’, ‘Maria’)	Validar que um valor é diferente do utilizado na comparação.
True	bool.should(‘be.true’)	Validar que um valor booleano é igual a true.
Exist	Button.should(‘exist’)	Validar que um elemento ou valor existe na tela.
lengthOf	‘Text’.should(‘have.lengthOf’, 4)	Validar o tamanho de um array ou texto.

Fonte: Elaboração própria.

4.3.6 Massa de dados

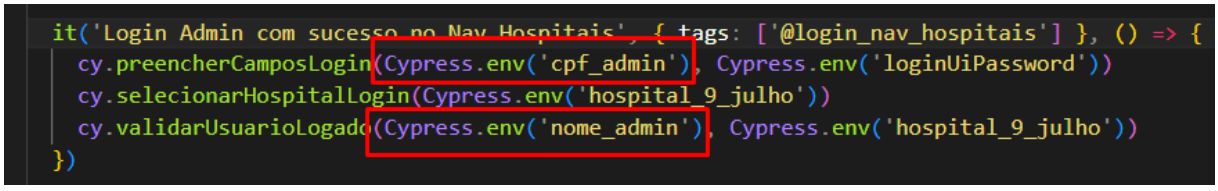
Com Cypress é possível encapsular toda a massa de dados utilizada em arquivos no formato JSON, agrupando assim todos os dados necessários para atender as entradas e saídas esperadas para os cenários de teste de uma forma simples e de fácil manutenção. Também é possível ter múltiplos arquivos para diferentes ambientes da aplicação, possibilitando assim de forma simples reaproveitar os testes em mais de um ambiente. Nas figuras 7 e 8 são demonstrados a estruturação da massa de dados e a utilização nos cenários de teste.

Figura 7 - Estrutura da massa de dados no formato JSON



Fonte: Elaboração própria.

Figura 8 - Utilização da massa de dados em cenário de teste.



Fonte: Elaboração própria.

4.3.7 Método It

Utilizando todos os conceitos citados anteriormente como commands, interações com o navegador, assertions e massa de dados, é possível criar um cenário de teste que reproduza o fluxo de interação do usuário com a aplicação e através desse cenário garantir que a aplicação se comporta como o esperado. Em Cypress, cada cenário de teste é escrito dentro de um método It, onde cada utilização do método corresponderá a um cenário dentro de uma determinada funcionalidade, como mostrado abaixo na Figura 9.

Figura 9 - Casos de testes separado em métodos it

```
it('Login utilizando CPF no campo de identificação', () => {
  cy.preencherCamposLogin(Cypress.env('usuario_cpf'), Cypress.env('loginUiPassword'))
  cy.selecionarHospitalLogin(Cypress.env('hospital_9_julho'))
  cy.validarUsuarioLogado(Cypress.env('nome_usuario_cpf'), Cypress.env('hospital_9_julho'))
})

it('Validar erro ao logar sem preencher email', () => {
  cy.loginSemPreencherEmail(Cypress.env('loginUiPassword'))
  cy.get(Cypress.env('login').p_mensagem_erro)
    .should('be.visible')
    .should('have.text', 'CPF inválido')
})

it('Validar erro ao logar sem preencher senha', () => {
  cy.loginSemPreencherSenha(Cypress.env('cpf_admin'))
  cy.get(Cypress.env('login').p_mensagem_erro)
    .should('be.visible')
    .should('have.text', 'Senha incorreta')
})
})
```

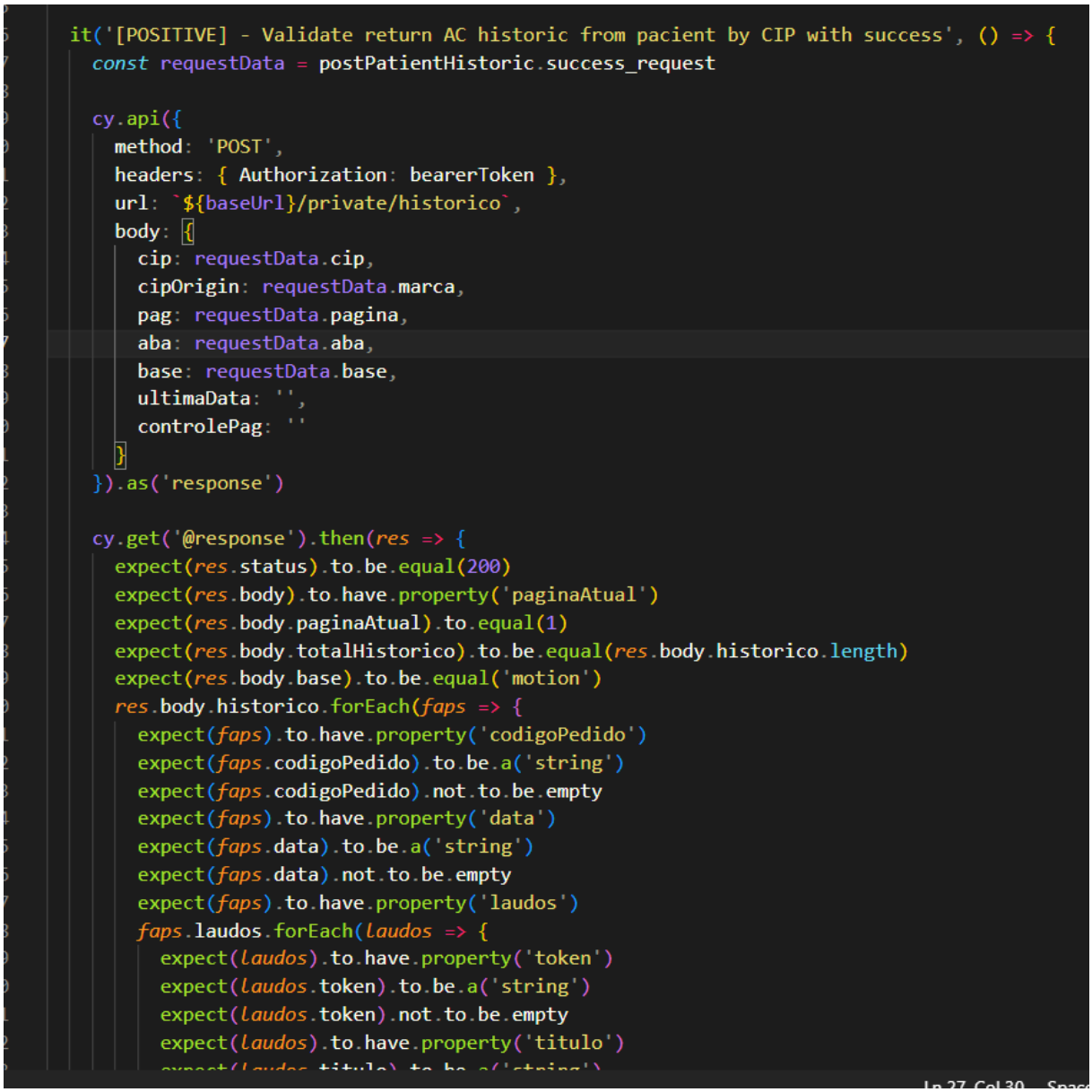
Fonte: Elaboração própria.

Cada uma das utilizações do método it indica um cenário de teste independente, que reproduz todo o fluxo de interação do usuário, desde as ações que devem ser efetuadas na aplicação, e as validações que garantirão que o comportamento está ocasionando o comportamento esperado.

4.4. Testes de API utilizando Cypress

Com Cypress também é possível nativamente escrever testes de API, permitindo assim que além de testes na interface gráfica, sejam feitos testes de serviços, podendo assim efetuar testes de contrato, integração, mock, regressão e end-to-end de forma extremamente rápida, por se tratar de serviços, utilizando de todas as demais vantagens do Cypress. Os testes de API são efetuados utilizando o método *cy.api*, onde serão parametrizados o método HTTP do serviço, os cabeçalhos da requisição, o recurso do serviço, e os demais requisitos do serviço, caso necessário. E por fim são feitas as validações necessárias através de asserts, como demonstrado na figura 10.

Figura 10- Exemplo de teste de API utilizando Cypress



```
it('[POSITIVE] - Validate return AC historic from pacient by CIP with success', () => {
  const requestData = postPatientHistoric.success_request

  cy.api({
    method: 'POST',
    headers: { Authorization: bearerToken },
    url: `${baseUrl}/private/historico`,
    body: {
      cip: requestData.cip,
      cipOrigin: requestData.marca,
      pag: requestData.pagina,
      aba: requestData.aba,
      base: requestData.base,
      ultimaData: '',
      controlePag: ''
    }
  }).as('response')

  cy.get('@response').then(res => {
    expect(res.status).to.be.equal(200)
    expect(res.body).to.have.property('paginaAtual')
    expect(res.body.paginaAtual).to.equal(1)
    expect(res.body.totalHistorico).to.be.equal(res.body.historico.length)
    expect(res.body.base).to.be.equal('motion')
    res.body.historico.forEach(faps => {
      expect(faps).to.have.property('codigoPedido')
      expect(faps.codigoPedido).to.be.a('string')
      expect(faps.codigoPedido).not.to.be.empty
      expect(faps).to.have.property('data')
      expect(faps.data).to.be.a('string')
      expect(faps.data).not.to.be.empty
      expect(faps).to.have.property('laudos')
      faps.laudos.forEach(laudos => {
        expect(laudos).to.have.property('token')
        expect(laudos.token).to.be.a('string')
        expect(laudos.token).not.to.be.empty
        expect(laudos).to.have.property('titulo')
```

Fonte: Elaboração própria.

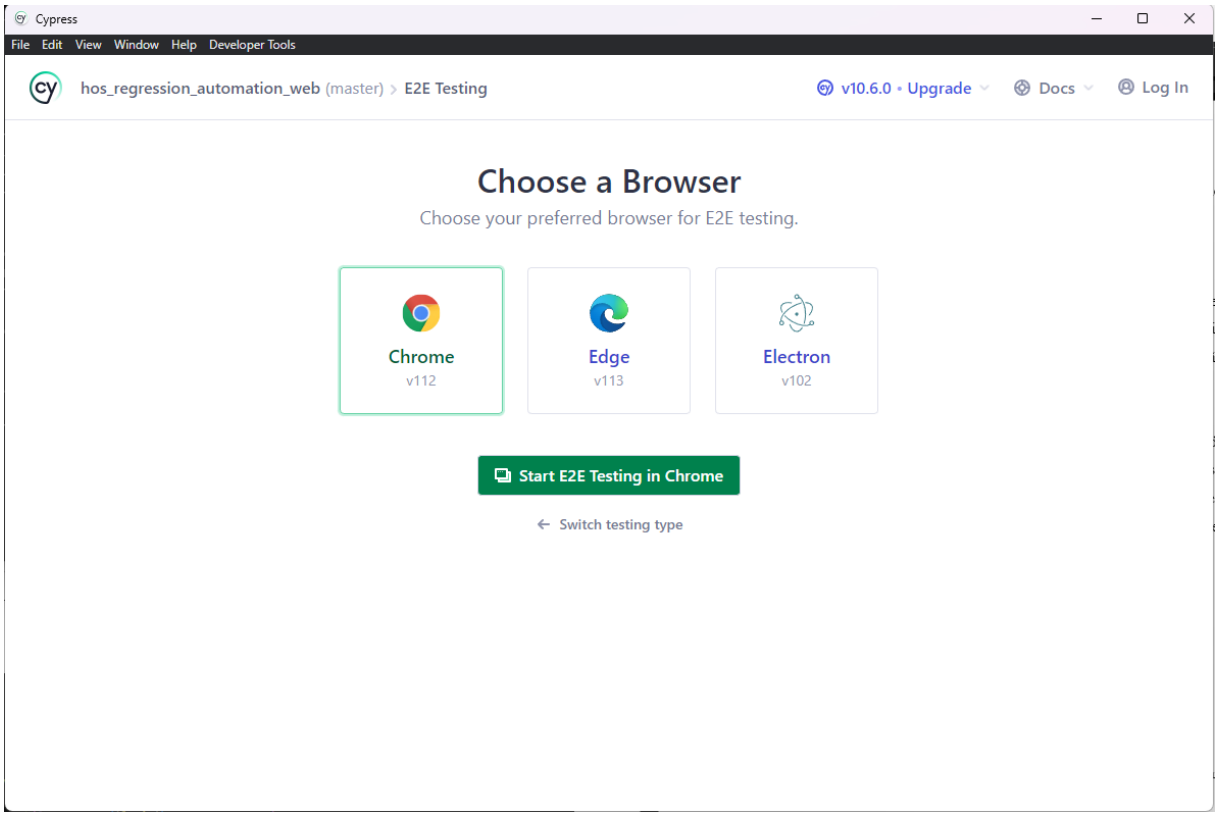
4.5. Execução dos scripts de testes automatizados

Uma das maiores vantagens do Cypress é a facilidade e eficiência de executar e compreender o fluxo dos testes sendo executados através de uma interface gráfica intuitiva e repleta de recursos para facilitar o desenvolvimento e manutenção dos testes. Neste tópico serão abordados alguns dos recursos disponibilizados pelo Cypress em modo de execução.

4.5.1 Execução em múltiplos navegadores

No Cypress é disponibilizado de forma nativa a execução de testes automatizados em múltiplos navegadores, o que para a garantia de qualidade através de testes automatizados é de suma importância, visto que a qualidade da aplicação e experiencia do usuário podem ser garantidas independente do navegador que o usuário utilize para acessar a aplicação. Na Figura 11 é demonstrada a interface de seleção de múltiplos navegadores, onde de maneira extremamente simples é possível selecionar o navegador no qual os testes serão executados.

Figura 11 - Seletor de diferentes navegadores na interface gráfica do Cypress



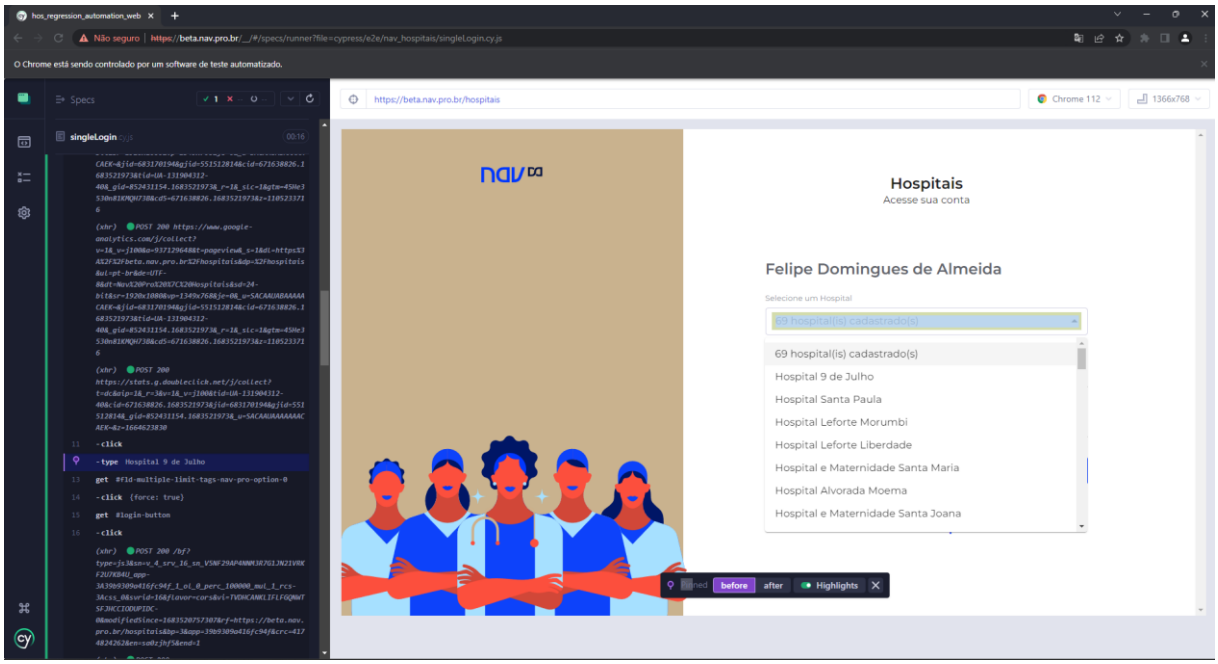
Fonte: Elaboração própria.

4.5.2 Cypress Time Travel

Na execução via interface gráfica do Cypress é possível utilizar uma função chamada de Time Travel, onde cada passo execução de um teste automatizado é gravado em um snapshot, ou seja, uma cópia do estado da aplicação durante toda a execução. Com essa funcionalidade é possível verificar o estado de cada passo do teste, antes e depois da interação com o navegador. Dessa forma se torna extremamente simples de entender erros em determinados passos do teste, e se eles foram causados por falha no desenvolvimento do script de teste ou por erro na aplicação. Sendo assim, é possível durante a própria execução dos testes

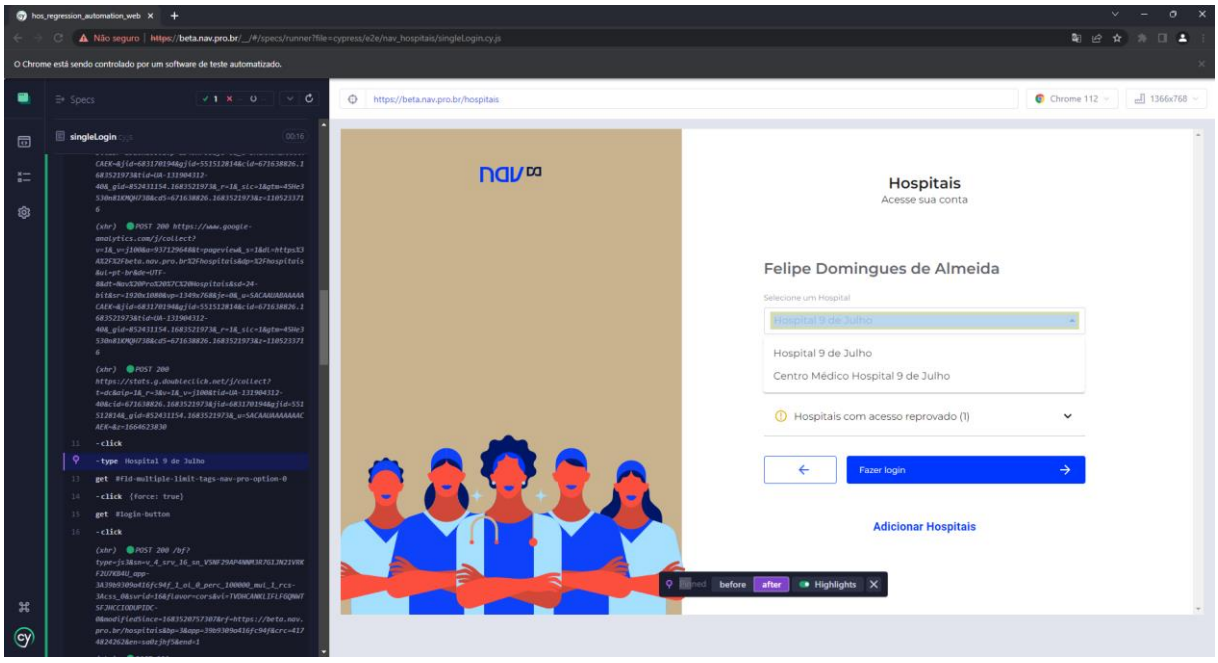
encontrar bugs o mais cedo possível e corrigi-los. Nas figuras 12 e 13 é demonstrada a utilização da funcionalidade Time Travel.

Figura 12 - Estado da aplicação antes de digitar um texto no campo



Fonte: Elaboração própria.

Figura 13 - Estado da aplicação após digitar um texto no campo

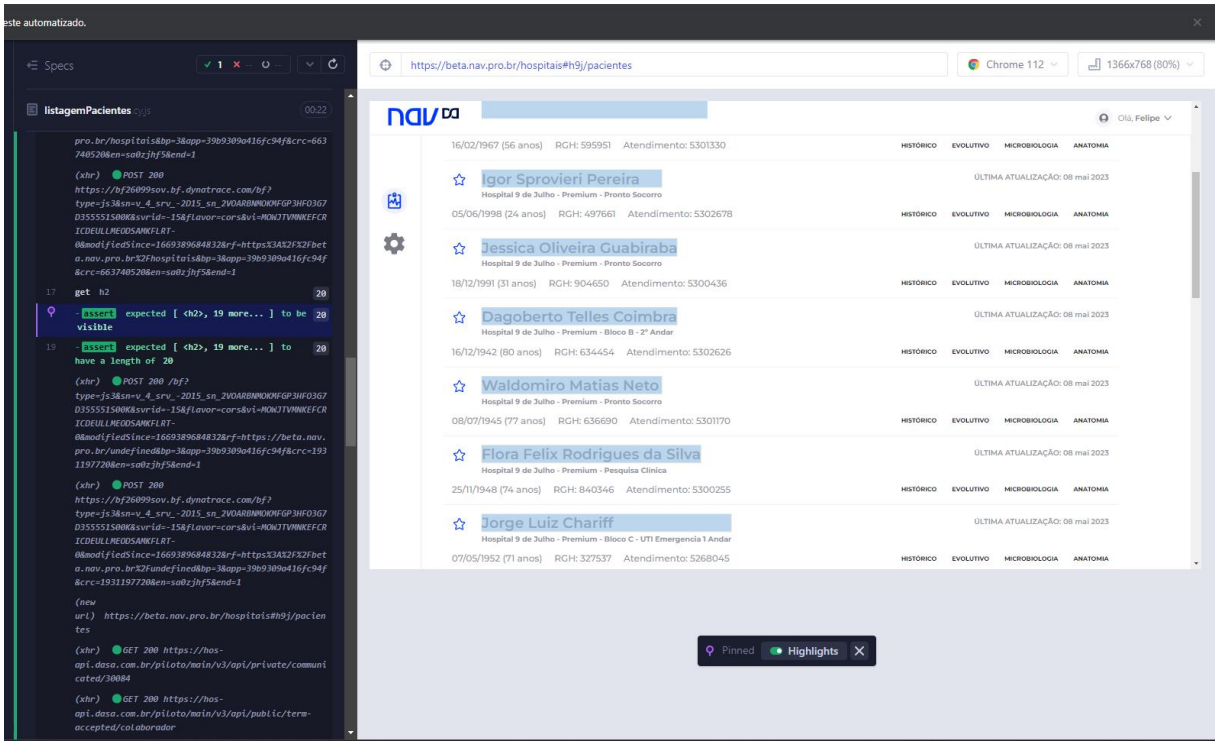


Fonte: Elaboração própria.

4.5.3 Timeline de execução dos testes

Mais uma vantagem relacionada a execução dos testes automatizados com Cypress é a possibilidade de acompanhar todas as ações do cenário de teste em uma linha do tempo em que são listados todos os comandos efetuados durante o teste, interações do script com a aplicação e as chamadas de back-end feitas pela aplicação ao longo do fluxo. Essa funcionalidade é de extrema importância para entender o comportamento do script de testes e da aplicação e se ambos estão de acordo com o resultado desejado. Também é possível clicar em cada um dos passos listados nessa linha do tempo para visualizar o estado da aplicação no exato momento em que o passo foi efetuado, conforme demonstrado na Figura 14.

Figura 14- Interação com a timeline com o passo a passo do teste



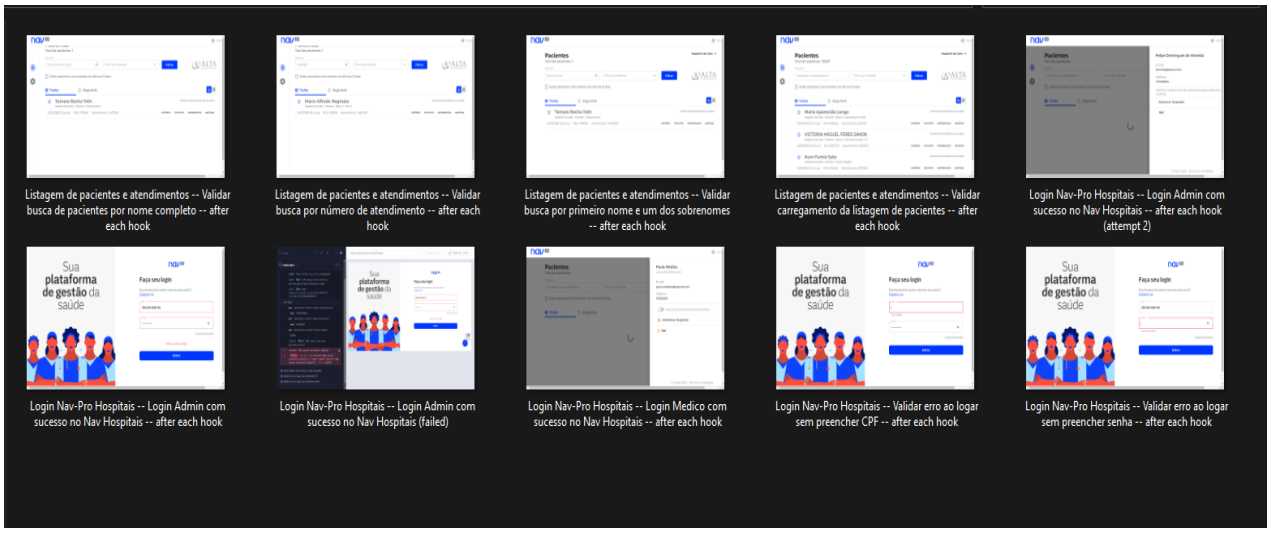
Fonte: Elaboração própria.

4.6. Evidências dos cenários de teste

Utilizando Cypress é possível configurar o framework para que gere evidências dos cenários de testes executados, tanto em caso de sucesso quanto em caso de falha. É possível gravar vídeos da execução de cada cenário e tirar capturas da tela em determinados pontos do teste, o que além de auxiliar na certificação de que o teste e a aplicação estão se comportando

como deveriam, em casos de erro é possível utilizar as evidências para entender com mais profundidade o cenário em que o erro ocorre, e caso seja necessária uma correção, compartilhar essas evidências com o time de desenvolvedores para facilitar o entendimento do problema. Na figura 15 é demonstrado um exemplo das capturas de tela geradas durante a execução de testes.

Figura 15- Exemplo de capturas de tela dos cenários de teste executados

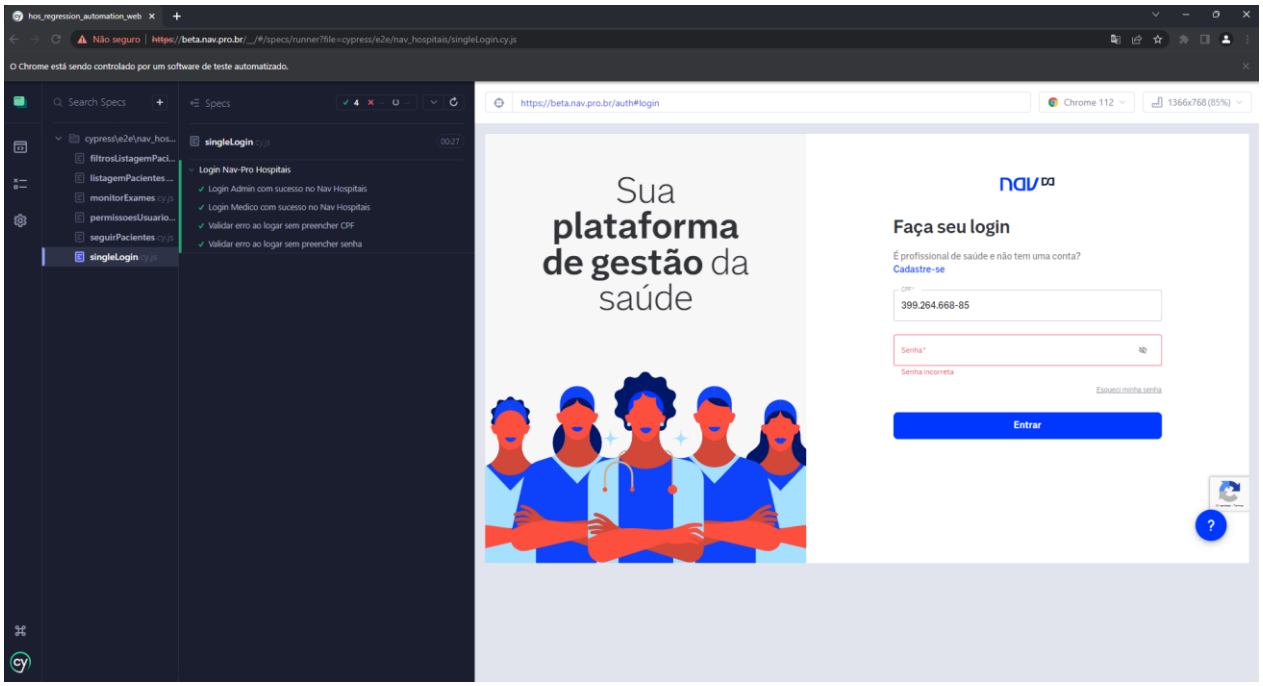


Fonte: Elaboração própria.

4.7. Execução do escopo de testes

Para finalizar a apresentação das vantagens de utilização do Cypress, foram desenvolvidos e executados todos os cenários de testes listados no escopo citados na Tabela 1. Os testes da funcionalidade “Login” foram executados em 27 segundos e todos foram executados com sucesso, conforme demonstrado na Figura 16.

Figura 16- Execução dos casos de testes da funcionalidade Login

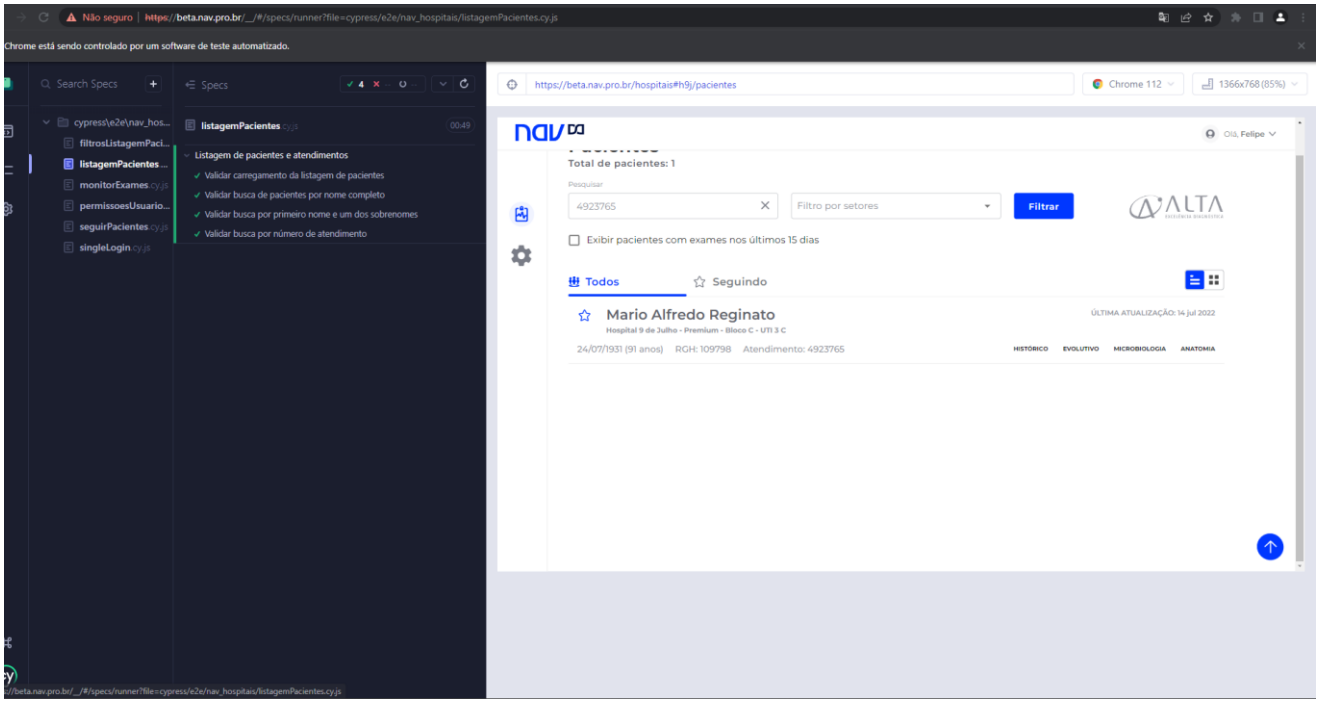


Fonte: Elaboração própria.

Já os testes da funcionalidade “Listagem de Pacientes” foram executados em 47 segundos e também foram executados com sucesso, conforme apresentado na Figura 17

.

Figura 17- Execução dos casos de testes da funcionalidade Listagem de Pacientes



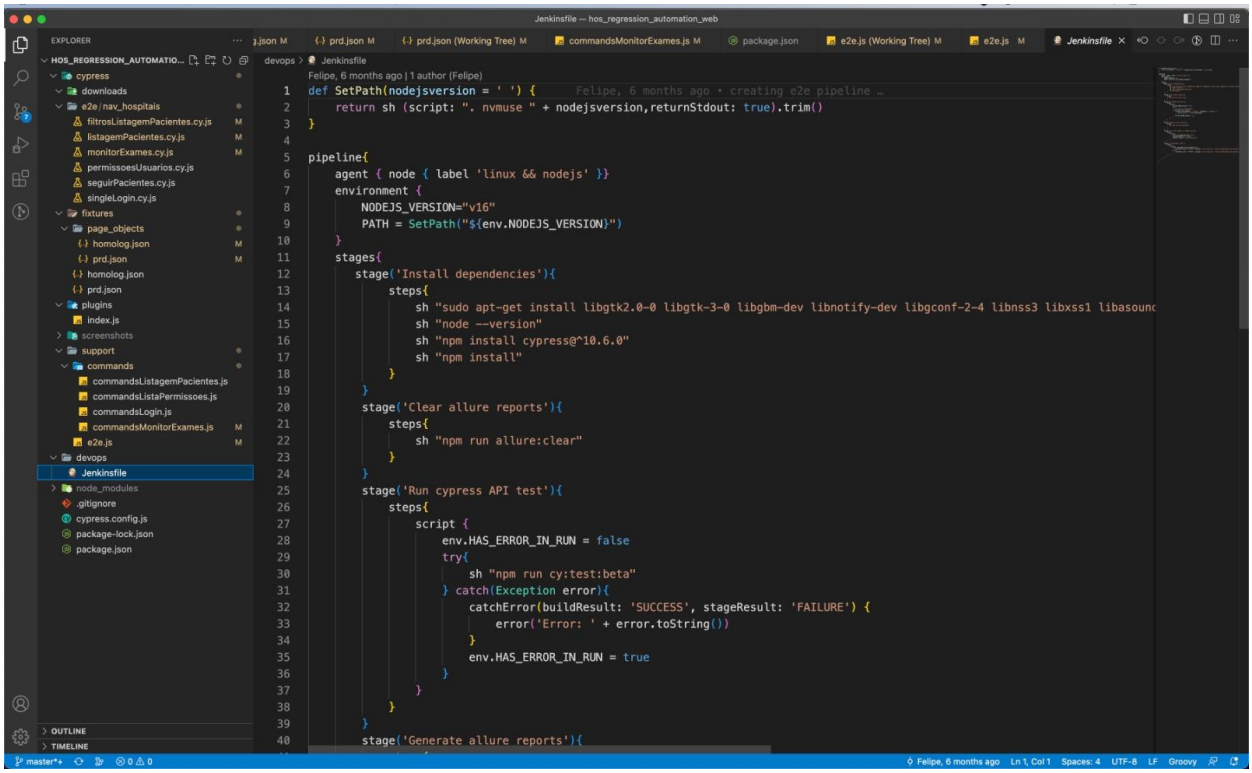
Fonte: Elaboração própria.

Mesmo utilizando um escopo limitado para demonstração, através dessas execuções é possível mensurar quantitativamente o quão eficiente e rápido o Cypress é para executar testes automatizados, e assim reduzir custos em execução de testes manuais, e tornar mais ágil o processo de desenvolvimento de software, garantindo testes executados de forma rápida e eficaz e consequentemente garantindo uma melhor qualidade do produto e experiencia do usuário.

4.8 Integração dos testes em uma pipeline utilizando Jenkins

Como citado no tópico 3.5, a integração continua permite automatizar tarefas visando tornar o processo de desenvolvimento de software mais ágil e confiável. Utilizando a ferramenta Jenkins, é possível criar uma pipeline para automatizar toda o fluxo de execução de testes automatizados, fazendo com que os testes sejam executados sem necessidade de uma pessoa para disparar essa execução. As configurações da esteira de testes são feitas através de um arquivo chamando *JenkinsFile*, onde serão especificados cada passo necessário para a execução dos testes automatizados, como as configurações do ambiente na nuvem, comandos para executar os testes, e criação do relatório de testes, por exemplo, como demonstrado na Figura 18.

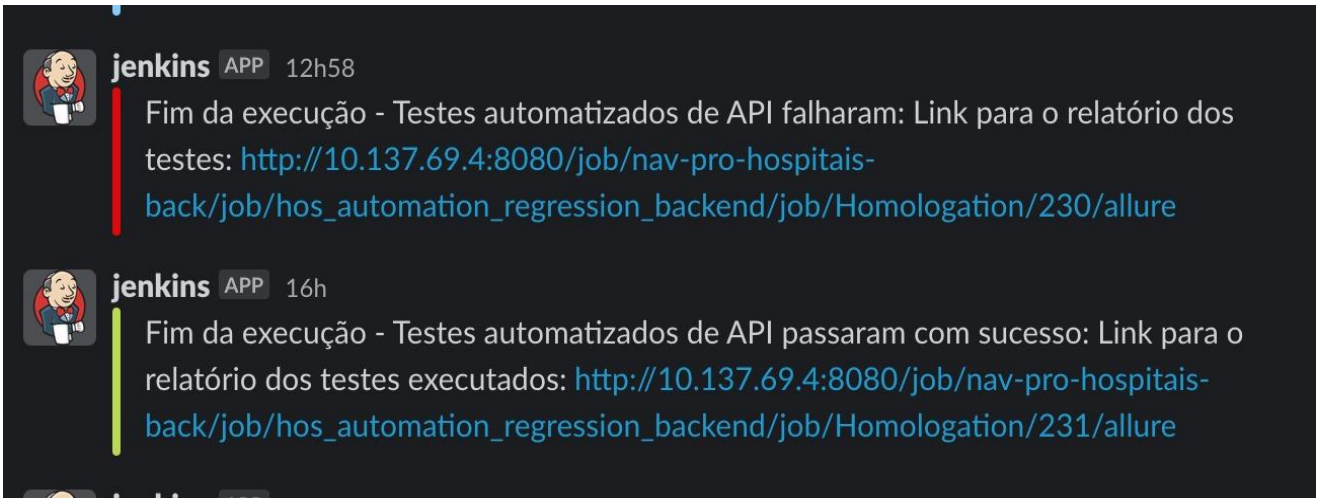
Figura 18 - Arquivo JenkinsFile contendo todos os passos a serem executados no pipeline de testes



Fonte: Elaboração própria.

Uma das principais vantagens da utilização de uma pipeline de testes é a possibilidade de integrar a esteira de testes com a esteira de desenvolvimento, possibilitando assim configurar o Jenkins para que cada vez que seja feita uma alteração no código da aplicação, a esteira de testes inicie automaticamente e execute os testes garantindo que as alterações efetuadas não quebraram alguma funcionalidade da aplicação, implicando que a garantia de qualidade será aplicada continuamente. Também é possível integrar a pipeline com softwares de comunicação corporativa, como o Slack ou Teams, fazendo com que seja disparada uma mensagem para o time com o status da execução dos testes, se falharam ou passaram, dando maior visibilidade do processo para os envolvidos no ciclo de desenvolvimento de software. Vide exemplo da integração na Figura 19.

Figura 19 - Integração da esteira de testes com ferramenta de comunicação Slack



Fonte: Elaboração própria.

4.9. Relatório de execução dos testes

Ao final de cada execução dos testes é possível gerar um relatório para compartilhar com o time um feedback detalhado de como se sucederam os testes da aplicação, dando assim visibilidade para a equipe e facilitando a compreensão e tomada de decisões de acordo com os resultados obtidos. Como o Cypress utiliza a linguagem JavaScript, é possível integrar os testes com a biblioteca cypress-allure-plugin, que permite gerar de forma automática um relatório que apresenta uma visão geral dos testes, fluxo detalhado de cada teste executado e gráficos da execução como um todo, como evidenciado nas figuras 20, 21 e 22.

Figura 20 - Visão geral do relatório de testes automatizados com Allure Reports

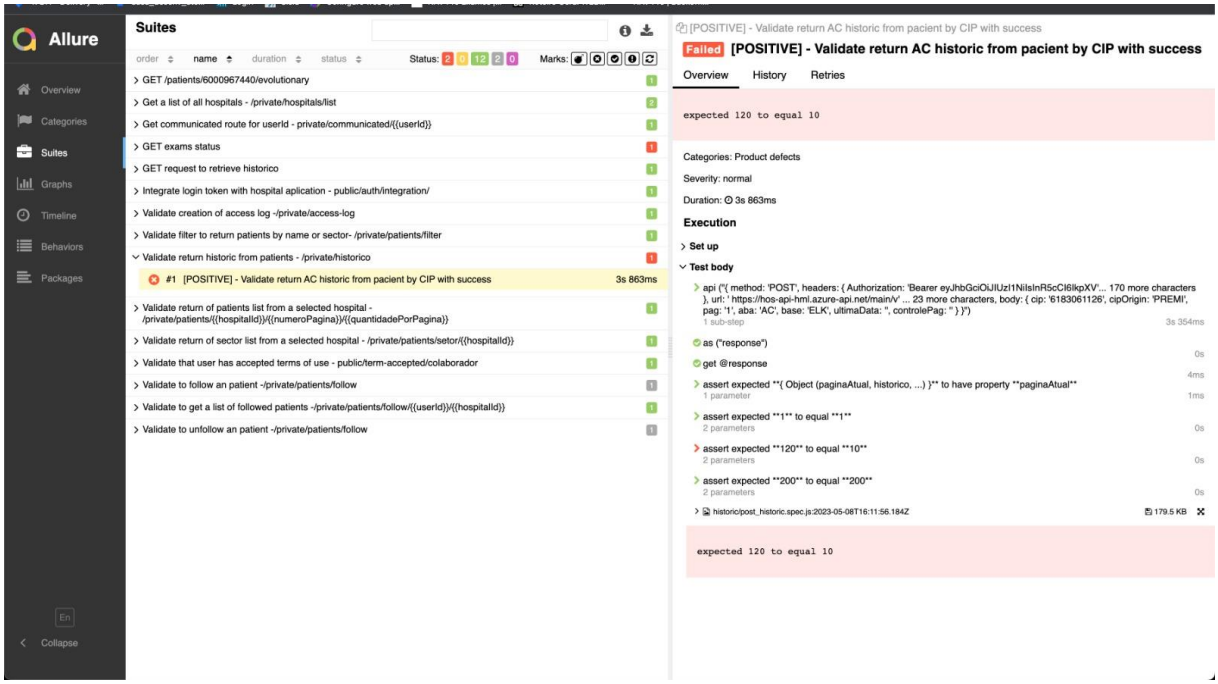
Fonte: Elaboração própria.

Nesse dashboard geral são apresentados dados da execução dos testes como a quantidade total de cenários executados, a porcentagem de testes que foram executados com sucesso, data da execução e o tempo total de execução dos testes. Também são apresentados dois gráficos, um de anel que representa a quantidade de testes executados e seus status divididos por cores, onde cada cor representa um status dos testes, sendo:

- Verde: testes executados com sucesso.
- Vermelho: testes que falharam na execução, pois as expectativas dos testes não foram atendidas.
- Cinza: testes ignorados, seja por estratégia de testes ou por outro motivo definido pelo desenvolvedor dos testes automatizados.
- Amarelo: testes que falharam, ou não foram executados, devido a problemas técnicos

Do lado direito do Dashboard é apresentado um gráfico de tendencia, esse gráfico é exibido quando a execução é feita através de uma pipeline de testes, onde ele representa o status dos testes ao longo das execuções, seguindo as cores citadas anteriormente para representar os status dos testes, dando assim uma visão geral de como os testes estão se comportando ao longo do tempo, possibilitando assim ao time focar em cenários que costumam apresentar mais problemas ao longo das execuções.

Figura 21 - Visão detalhada dos testes automatizados no relatório



Fonte: Elaboração própria.

Na visão detalhada é exibida uma lista de todos os testes executados, o status do cenário na execução, e ao clicar no nome do cenário é aberta uma área com detalhes do teste, todo o fluxo executado no cenário, quais foram as ações efetuadas, e as comparações efetuadas para validar o teste, mostrando qual o valor recebido e qual o valor esperado. Também são anexadas as capturas de tela configuradas ao longo dos cenários.

Figura 22 - Visão de gráficos do relatório de testes automatizados

Fonte: Elaboração própria.

Por fim, na visão de gráficos são apresentados diversos gráficos para dar maior visibilidade da execução e histórico de execuções como um todo. Na imagem são apresentados um gráfico de anel, onde é detalhada a porcentagem e status dos cenários executados. Um gráfico de barras, onde é possível configurar diferentes severidades para cada testes, e assim exibir os cenários de acordo com a severidade definida. Um gráfico de barras mostrando o tempo médio de duração de cada cenário durante a execução, e um gráfico mostrando o histórico de duração da execução de toda a suíte de testes na pipeline ao longo do tempo.

CONSIDERAÇÕES FINAIS

Ao concluir este trabalho sobre testes automatizados utilizando o framework Cypress, foi possível compreender a importância da automação de testes no processo de desenvolvimento de software, bem como a facilidade e a eficiência que o Cypress oferece para essa tarefa.

Nesse sentido, o framework Cypress se destaca por sua simplicidade, facilidade de uso e ampla documentação. Sua arquitetura baseada em JavaScript permite a criação de testes de forma intuitiva e ágil, além de oferecer diversas funcionalidades e ferramentas para a criação de testes complexos e robustos.

Durante a realização deste trabalho, foram explorados diversos recursos do Cypress, como a criação de testes de serviço e End-to-end, a simulação de interações do usuário com o sistema e a execução de testes em diferentes navegadores e dispositivos.

Além disso, também foi possível compreender a importância da integração contínua no processo de desenvolvimento de software, que consiste na execução automatizada de testes a cada nova versão ou atualização do sistema. Essa prática permite a detecção rápida e eficiente de possíveis problemas e garante que o software esteja sempre funcionando de forma adequada.

Outro ponto importante abordado neste trabalho foi a importância da documentação dos testes automatizados, que permite a fácil compreensão e manutenção dos mesmos. O Cypress oferece diversas ferramentas e funcionalidades para a criação de uma documentação clara e objetiva, o que facilita o trabalho dos desenvolvedores e testadores.

Por fim, é importante destacar que a utilização do framework Cypress pode trazer diversos benefícios para as empresas e equipes de desenvolvimento de software, como a redução de custos e tempo de desenvolvimento, a melhoria da qualidade do software e a satisfação do usuário final.

Em resumo, este trabalho permitiu compreender a importância dos testes automatizados e da utilização do framework Cypress, bem como os benefícios que essa prática pode trazer para as empresas e equipes de desenvolvimento de software. Dessa forma, recomenda-se fortemente a utilização do Cypress como ferramenta de automação de testes, visando garantir a qualidade e eficiência do software desenvolvido.

REFERÊNCIAS

BASTOS, A.; RIOS, E. CRISTALLI, R. & MOREIRA, T. Base de conhecimento em teste de software. São Paulo, Martins Fontes, 2007.

BEIZER, B. Software Testing Techniques. 2nd edition. New York: Van Nostrand Reinhold, 1995.

BOAGLIO, Fernando. Jenkins. Automatize tudo sem complicações. Casa do Código, 2016.

CRISPIN, L.; GREGORY, J. Agile Testing: A Practical Guide for Testers and Agile Teams. Pearson Education, 2009.

COHN, Mike. Succeding with Agile: Software Development Using Scrum. 1. Ed. Addison Wesley Professional, 2009.

CYPRESS. The Story of Cypress.io. 2022. Disponível em: https://www.cypress.io/about-us/our-story. Acesso em: 13 mai. 2023.

DUVALL, Paul; MATOUSEK, Steve; HUMBLE, Jez. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007.

FLANAGAN, David. JavaScript: The Definitive Guide. O'Reilly Media, 2020.

FOWLER, M. Testes Unitários, 2014. Disponível em: https://martinfowler.com/bliki/UnitTest.html. Acesso em: 03 abr. 2023.

ISTQB, International Software Testing Qualifications Board. Foundation Level Syllabus – BSQTB, 2018.

MARTIN, R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2013.

MOLINARI, Leonardo. Inovação e automação de testes de software. 1. Ed. São Paulo: Erica, 2010.

MOLINARI, Leonardo. Testes de Software: Produzindo Sistemas melhores e mais confiáveis. 4. ed. São Paulo: Erica, 2008.

MUSTAFA, K.; KHAN, R. Software Testing: Concepts and Practices. Alpha Science International, 2007

MYERS, G. J., Badgett, T., & Sandler, C. (2011). The art of software testing. 3ª ed. New Jersey: John Wiley & Sons.

NOGUEIRA, Elias. Testando microservices com Rest-Assured. 2020.

PRESSMAN, R. S. Engenharia de Software. 8ª ed. São Paulo: McGraw Hill, 2016.

RIOS, Emerson; MOREIRA, Trayahú. Teste de Software. 3ª. ed. rev. e aum. Rio de Janeiro: Alta Books, 2013.

SOMMERVILLE, I. Engenharia de Software. 9ª ed. São Paulo: Pearson Education, 2011.