Testes automatizados em interfaces web

Hunter 2.0

Marcos Davi Nascimento

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2024

Marcos Davi Nascimento

Testes automatizados em interfaces web

Monografia apresentada ao curso Engenharia de Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Thaís Gaudencio do Rêgo

Novembro de 2024

Catalogação na publicação
Seção de Catalogação e Classificação

N244t Nascimento, Marcos Davi.
Testes automatizados em interfaces web : Hunter 2.0 / Marcos Davi Nascimento. - João Pessoa, 2024.
49 f. : il.

Orientação: Thaís Gaudencio Rêgo.
Coorientação: Yuri Malheiros, Yuska Paola Costa Aguiar.
TCC (Graduação) - UFPB/CI.

1. Laboratório ARIA. 2. Secretaria da fazenda. 3. rotina de testes. 4. desenvolvimento de ferramenta. 5. análise de tributação fiscal. 6. Jest. I. Rêgo, Thaís Gaudencio. II. Malheiros, Yuri. III. Aguiar, Yuska Paola Costa. IV. Título.

UFPB/CI CDU 004.774

Elaborado por Michelle de Kassia Fonseca Barbosa - CRB-738

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Engenharia de Computação intitulado ***Testes automatizados em interfaces web*** de autoria de Marcos Davi Nascimento, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dra. Thaís Gaudencio
UFPB

Prof. Dr. Yuri Malheiros
UFPB

Prof. Dra. Yuska Paola Costa Aguiar
UFPB

João Pessoa, 29 de novembro de 2024

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

AGRADECIMENTOS

Primeiramente, quero agradecer aos meus pais, Erminio e Luzinete, que sempre me apoiaram nas minhas escolhas, me preparam para o ambiente acadêmico e me fizeram acreditar que sou capaz de atingir qualquer um dos meus objetivos.

Agradeço também a minha irmã Gabriela, que esteve comigo durante toda esta jornada, em uma cidade distante e sendo o único membro da família sempre ao meu lado.

Agradeço aos professores e orientadores que tive durante o curso, principalmente Thaís Gaudencio e Yuri Malheiros, que me guiaram desde o ínicio da graduação e não me deixaram desistir perante as dificuldades. Sem eles, não teria sido capaz de concluir minha formação, apesar de todo o apoio que recebi de amigos e família.

RESUMO

O Laboratório ARIA, em parceria com a Secretaria da Fazenda do Estado da Paraíba, estuda o desenvolvimento de uma ferramenta robusta, que visa facilitar o processo de análise de tributação fiscal de empresas que atuam no Estado da Paraíba. Com base neste software, o presente trabalho visa implementar uma rotina de teste, para garantir o funcionamento desejado da ferramenta, validando seu comportamento padrão, fluxos alternativos e de exceção e suas respostas à interações feitas pelos usuários. A implementação da solução foi direcionado à tela "Explorar dados", que possui um vasto número de dados e operações disponíveis. Os casos de testes foram implementados utilizando de bibliotecas mais comuns para validação de projetos desenvolvidos em React e TypeScript, como a biblioteca Jest e Testing-library/react.

Palavras-chave: Laboratório ARIA, Secretaria da Fazenda, rotina de testes, desenvolvimento de ferramenta, análise de tributação fiscal, Jest.

ABSTRACT

The ARIA Laboratory, in partnership with the State Department of Finance of Paraíba, is studying the development of a robust tool aimed at facilitating the process of fiscal taxation analysis for companies operating in the state. Based on this software, the present work aims to implement a testing routine to ensure the desired functioning of the tool, validating its standard behavior and its responses to user interactions. The implementation of the solution was directed towards the "Explore Data"screen, which has a vast number of available data and operations. The test cases were implemented using the most common libraries for validating projects developed in React and TypeScript, such as the Jest library and Testing-library/react.

Key-words: ARIA Laboratory, State Department of Finance of Paraíba, testing routine, development of a robust tool, fiscal taxation analysis, Jest.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ARIA – Artificial Intelligence Applications laboratory

CNPJ - Cadastro Nacional da Pessoa Jurídica

DOM - *Document Object Model*

HTML - *HyperText Markup Language*

JS - JavaScript

NF-e - Nota Fiscal Eletrônica

UFPB - Universidade Federal da Paraíba

TS - TypeScript

XML - *Extensible Markup Language*

Sumário

1 INTRODUÇÃO

A Secretaria da Fazenda da Paraíba desempenha um papel fundamental na administração tributária e no controle das finanças públicas do Estado. Para realizar a fiscalização tributária de pessoas jurídicas, um auditor fiscal precisa ter acesso a centenas de campos de preenchimento de notas fiscais e relatórios de estoque gerados por empresas. Isto representa um enorme número de dados, o que dificulta o trabalho, atualmente feito manualmente, e pode levar a erros causados por falha humana.

Com o objetivo de otimizar esse processo, foi desenvolvida uma ferramenta *web* em uma parceria entre o Laboratório ARIA da UFPB e a Secretaria da Fazenda da Paraíba, que gerencia a visualização e *download* de dados de notas fiscais emitidas por CNPJs e também automatiza parte do processo de cruzamento destas informações, utilizando auxílio de uma Inteligência Artificial.

O sistema oferece três funcionalidades principais: o explorador, que permite ao auditor consultar e baixar dados das notas fiscais, acessados por meio da seleção de colunas e aplicação de filtros de busca; a tela de cruzamento, na qual o usuário pode utilizar a ferramenta para validar as informações emitidas por uma pessoa jurídica, em um período de tempo escolhido; e a tela de resultados, que exibe a lista de *downloads* e cruzamentos disparados pelo usuário, permitindo também acessar os dados obtidos pelo algoritmo do Hunter.

1.1 Definição do Problema

Assim como qualquer software, é fundamental assegurar que a ferramenta funcione corretamente, principalmente quando consideramos seu propósito. Porém, atualizações de códigos e alterações de requisitos podem frequentemente introduzir falhas inesperadas no sistema, muitas vezes difíceis de detectar. Com este cenário, fez-se necessária a utilização de testes automatizados, para garantir que cada parte da aplicação demonstre o comportamento esperado ao longo de todas as etapas do desenvolvimento, diminuindo riscos e melhorando a qualidade do produto final.

Os testes são parte crucial do desenvolvimento de qualquer software, podendo ser implementados em diversas etapas e níveis diferentes durante a construção do produto. Os diferentes níveis de testes implementados (unitários, integração e ponta a ponta) visam garantir a qualidade do sistema, validando desde as ações mais simples até o funcionamento das partes como um todo.

1.2 Premissas e Hipóteses

A implementação de rotinas de testes automatizados de software irá tornar o desenvolvimento do sistema muito mais confiável e escalável, reduzindo o número de falhas causadas por mudanças e garantindo o funcionamento correto. Em adicional, os arquivos de teste funcionarão também como uma documentação da aplicação, descrevendo seu comportamento ideal e garantindo que isso seja mantido a cada nova versão lançada.

Até então, não existem dados de tempo utilizado para correção e número total de falhas encontradas durante o desenvolvimento. Isto por si só, já se classifica como um grande aspecto a ser melhorado na construção da ferramenta. A rotina de testes, diminuirá o tempo investido na procura e solução destas falhas e ajudará os desenvolvedores a entender rapidamente onde se encontra o problema e como corrigi-lo, validando também a nova solução, tendo em vista que os testes serão executados sempre que uma nova alteração é inserida no código.

1.3 Objetivo geral

Implementar uma rotina de testes automatizados, executada sempre que uma nova alteração é inserida na ferramenta e também quando for enviada para o ambiente final de produção, com intuito de garantir a qualidade, confiabilidade e eficiência da ferramenta, reduzindo erros e facilitando o processo de desenvolvimento e manutenção. Além disso, essa rotina também contribuirá para prevenir regressões[1] em versões futuras e atua como uma documentação do comportamento esperado do sistema, permitindo evolução contínua com mais segurança e controle.

1.4 Objetivos específicos

- Escolher ferramentas de testes adequadas.
- Documentar o funcionamento correto dos componentes do explorador.
- Implementar testes funcionais unitários para cada componente isolado da interface.
- Desenvolver testes de integração das partes do sistema.
- Desenvolver testes de validação de ponta a ponta, assegurando comportamento correto através das ações do usuário no fluxo de utilização do sistema.
- Elaborar testes de integração com a API de dados.

[1] Entende-se como regressões de software quando uma alteração de código, seja ela uma adição ou atualização de um recurso disponível, refatorações de código ou correções acabam gerando alterações inesperadas no comportamento geral do sistema.

- Implementar uma rotina de execução de testes durante o desenvolvimento da ferramenta.
- Garantir o funcionamento de novas partes incluídas no software e também assegurar o comportamento anterior à mudança.

1.5 Estrutura do relatório técnico

O trabalho está dividido em seis capítulos. O primeiro capítulo tem como objetivo apresentar uma introdução do produto e descrever a problemática abordada. O segundo capítulo descreve brevemente o funcionamento esperado da ferramenta de busca do explorador de dados. O terceiro capítulo possui um referencial teórico para conceitos e tecnologias utilizadas no desenvolvimento do produto e da solução. No quarto capítulo, é descrita a metodologia utilizada, explicando uma visão geral sobre a solução, uma classificação para os testes desenvolvidos, passos e conceitos necessários para a implementação, e como é alcançado o resultado esperado. O quinto capítulo visa analisar e discutir resultados alcançados através da metodologia definida. Por fim, o último capítulo contém as conclusões e considerações para trabalhos futuros.

2 HUNTER 2.0

O Hunter 2.0 é uma ferramenta em desenvolvimento pelo Laboratório ARIA da UFPB, que visa auxiliar auditores fiscais da secretária da fazenda a realizar a fiscalização de documentos fiscais emitidos. A base de dados da receita federal tem um volume significativo de dados, contendo informações emitidas em todas as regiões do Estado da Paraíba, e considerando este contexto, a aplicação foi desenvolvida para facilitar essa análise.

O processo de avaliação da tributação fiscal é feito através da análise de todos os documentos fiscais emitidos por uma pessoa jurídica. Ao todo, são mais de quatrocentos campos de informação em cada documento da base de dados que podem ser avaliados.

As seções a seguir explicam o funcionamento da tela "Explorar Dados", e seus principais componentes.

2.1 Explorar Dados

O objetivo da tela "Explorar Dados", exibida na Figura 1, é permitir que o auditor fiscal possa visualizar e fazer o *download* dos dados emitidos pelas pessoas jurídicas, facilitando consultas rápidas e permitindo realização do processo de avaliação das informações de maneira personalizada.

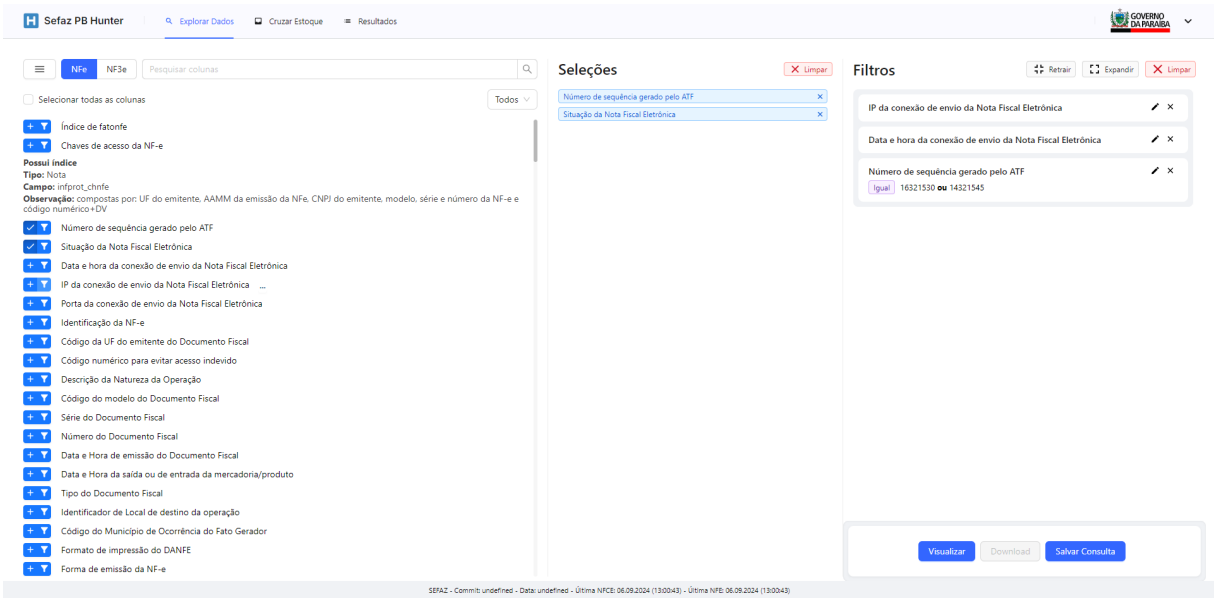


Figura 1: Explorador de dados
Fonte: própria do autor

2.1.1 Carregamento dos dados

Após o processo de *login* e autenticação, a aplicação redireciona o usuário para a página do explorador e faz quatro requisições para o *back-end*. O primerio *endpoint* acessado é o */description*, que retorna uma lista de todos os campos disponíveis para consulta. A segunda é uma outra requisição do tipo GET enviada para */saved-queries*, que irá retornar a lista de consultas configuradas e salvas pelo usuário logado. Assim que esses dados são recebidos do servidor, eles são armazenados no estado global da aplicação e gerenciados pelo Redux.

Com esses dados no lado do cliente, o usuário pode começar a fazer suas consultas de documentos fiscais, preenchendo os valores dos filtros escolhidos e selecionando os campos para visualização. Os campos selecionados e os filtros também são mantidos no Redux e atualizados, por meio de *actions* [2] disparadas por componentes da tela.

2.1.2 Organização

Considerando o grande número de dados, o conteúdo desta tela foi distribuído em três partes: menu lateral; conteúdo principal e visualização de filtros. Com esta configuração, o auditor pode personalizar quais dados deseja visualizar e aplicar filtros para a busca de informações no banco de dados.

1. **Menu Lateral**: fica oculto e pode ser acessado através de um clique no botão à esquerda da barra de pesquisa, nele é exibida a lista de consultas pré-configuradas pelo sistema e as consultas definidas e salvas pelo usuário. Ao selecionar um item, as colunas e filtros configurados podem ser reutilizados para consultas. Esta opção permite que o auditor faça a análise de um conjunto de dados de maneira rápida e simples.

2. **Conteúdo Principal**: Na seção principal, são listados todos os campos de informações disponíveis em um documento fiscal. Cada item da lista possui duas ações possíveis, a primeira é selecionar o campo para visualização, e a segunda é criar um filtro, referente àquele campo, que será utilizado na requisição das notas fiscais. É possível visualizar a descrição da informação que está contida naquele campo ao clicar no botão de informações, que aparece ao passar o mouse sobre o item.

3. **Visualização de Filtros**: A terceira e última parte é dividida em duas colunas: seleções e filtros. A primeira exibe a lista de campos de informações adicionados para

[2]Disponível em: https://redux.js.org/tutorials/essentials/part-1-overview-concepts. Acessado em 24 de setembro de 2024

visualização, permitindo que o usuário organize a ordem das colunas. A segunda coluna mostra ao usuário os campos selecionados para filtros, para que ele possa preencher os valores com diferentes combinações.

Na base da coluna, são exibidas as operações de consultas disponíveis no explorador de dados. São três operações, no total, que aparecem como botões da interface e fazem requisições para o *back-end*. O botão "Visualizar" faz uma pré-visualização dos dados encontrados no banco de dados, utilizando os filtros adicionados, exibindo somente os cem primeiros resultados em uma tabela. A opção "*Download*" busca todos os dados encontrados com os filtros, e gera um arquivo no formato desejado, que pode ser baixado posteriormente para uma visualização completa. Clicando no botão "Salvar Consulta", o sistema salva a seleção e os filtros atuais como uma consulta configurada, permitindo sua reutilização.

2.2 Componentes

Nesta seção, serão apresentados os principais componentes que são exibidos na interface do explorador de dados.

2.2.1 *ColumnBox*

Cada item da lista do conteúdo principal é exibido como um componente *ColumnBox*, como exibido na Figura 2, e representam os campos de informações de documentos fiscais. Nele são exibidos: botões para adicionar o campo para visualizar na consulta e adicionar como filtro; nome do campo e botão que exibe todas as informações relacionadas àquela coluna (descrição, código do campo, observação).

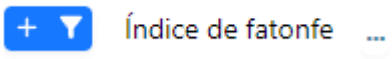


Figura 2: *ColumnBox*
Fonte: própria do autor

2.2.2 Filtros

O componente de filtro pode ser preenchido com configurações diferentes, permitindo buscas mais robustas e seletivas. Ele possui um campo de digitação para inserir o valor desejado na busca, podendo também aplicar algumas outras opções disponíveis no *checkboxes* na parte inferior.

- **Ou**: a opção "ou" exibe outro campo de texto para que o usuário insira um segundo valor para ser usado na busca;

- **Não**: com a opção “não”, a consulta irá retornar os documentos fiscais com valores diferentes do inserido no componente;
- **Intervalo**: a opção “intervalo” exibe dois campos de texto, referentes ao valor de início e fim do intervalo, que serão usados na consulta.

O sistema permite que o usário faça combinações de opções e preencha os campos de diferentes formas para fazer buscas mais especifícas na base de dados. A Figura 3 demonstra um exemplo de uso dos filtros. O primeiro procura por valores contidos no intervalo ou iguais ao segundo campo, e o último filtro busca valores diferentes dos dois inseridos.

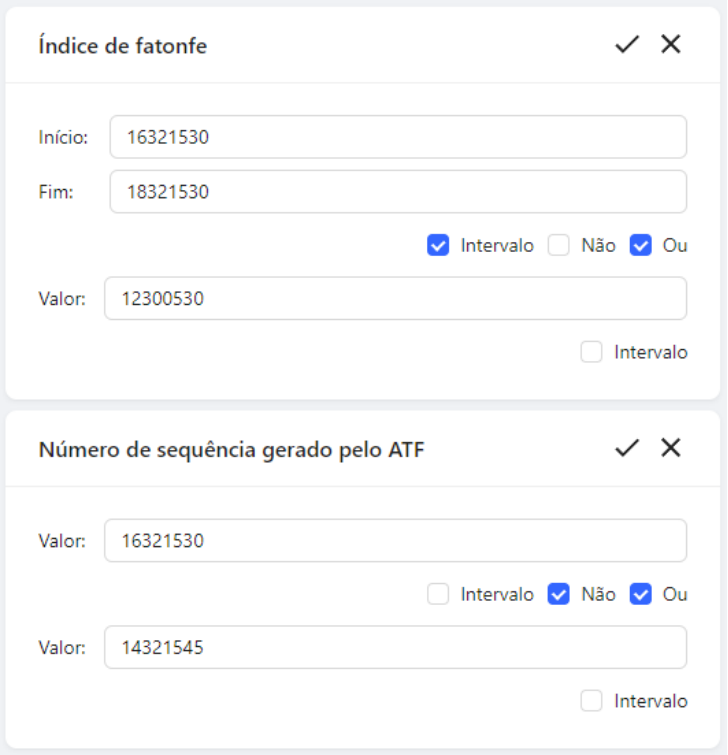


Figura 3: Filtros preenchidos
. Fonte: própria do autor

2.2.3 *Download*

O botão de *download* fica desabilitado até que o usuário inclua um filtro de data, ou uma chave de acesso de nota fiscal, para evitar consultas muito extensas no banco de dados. Ao ser pressionado, ele exibe um *pop-up*, ilustrado na Figura 4, para que o usuário preencha informações referentes ao arquivo que será gerado (nome, separador, extensão e exibição das colunas). Confirmando o *download*, a interface faz uma requisição para o *back-end* fazer uma busca de notas fiscais emitidas, aplicando os filtros adicionados.

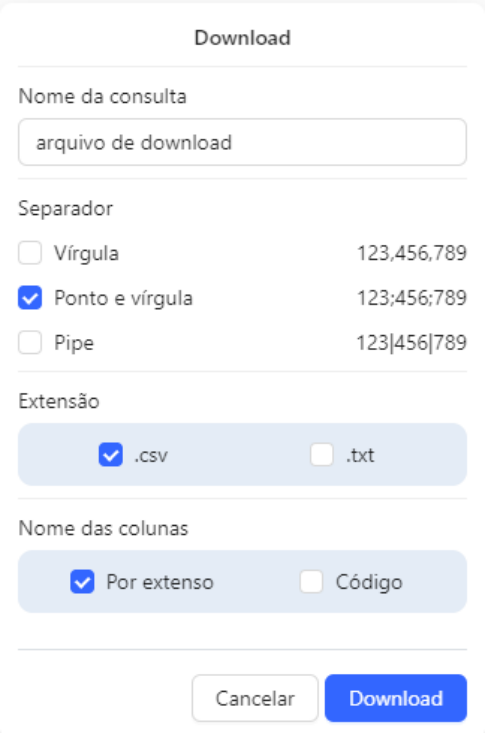


Figura 4: *Pop-up* de *Download*
. Fonte: própria do autor

2.2.4 Salvar Consulta

O componente de salvar consulta também é um *pop-up*, como mostra a Figura 5, e pode ser visualizado clicando no botão "Salvar Consulta" na base da coluna de filtros. Ele exibe um resumo da consulta que será salva e permite que o auditor insira o nome, que será utilizado para essa consulta. Ao salvar, a interface envia as colunas selecionadas e os filtros adicionados para o servidor manter essa informações no banco de dados, permitindo a reutilização desta configuração com um clique rápido no menu lateral.

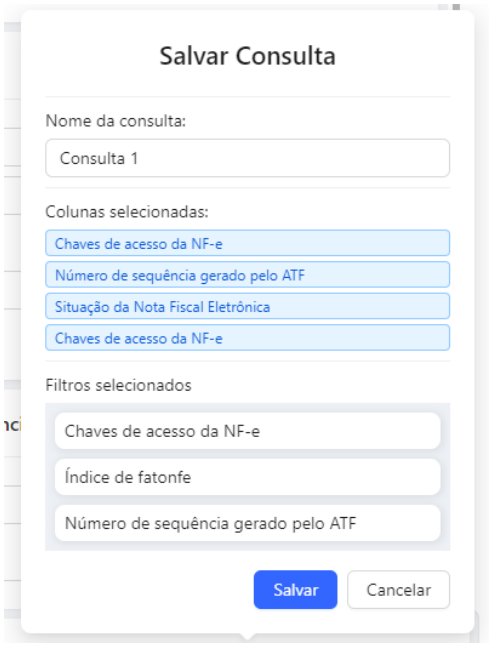


Figura 5: *Pop-up* de Salvar Consulta
. Fonte: própria do autor

2.2.5 *Preset*

O componente *Preset* fica no menu lateral e exibe a lista de consultas personalizadas pelo auditor e pré-configuradas pelo sistema. Ao selecionar um item desta lista, os filtros e colunas configurados são adicionados no estado global, para que possam ser utilizados na consulta. As consultas salvas pelo usuário podem ser removidas pressionando o botão "x" que fica visível ao posicionar o mouse no botão. A Figura 6 exemplifica como esses dados são exibidos para o auditor.

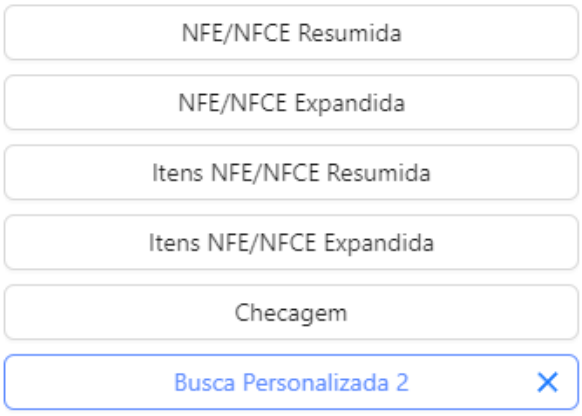


Figura 6: Consultas pré-configuradas
Fonte: própria do autor

3 CONCEITOS E TECNOLOGIAS

Neste capítulo serão introduzidos alguns conceitos necessários para o entendimento do funcionamento do produto e da solução.

3.1 Arquitetura da ferramenta

A Figura 7 mostra a arquitetura da ferramenta web Hunter 2.0 desenvolvida para facilitar o processo de auditoria feito pelos fiscais da Secretária da Fazenda do Estado da Paraíba.

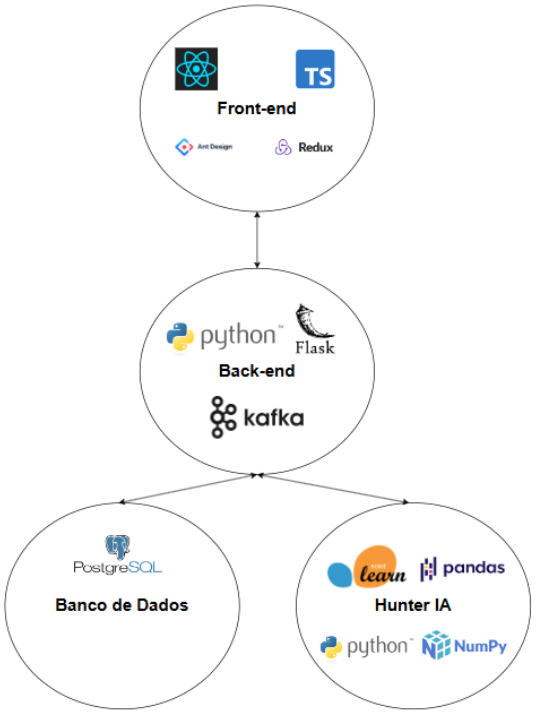


Figura 7: Arquitetura da ferramenta

3.2 Conceitos Gerais

Nesta seção, estão especificados alguns do conceitos principais para o funcionamento da ferramenta desenvolvida, explicando o conceito geral dos dois módulos principais (*front-end* e *back-end*) e dando foco nas ferramentas utilizadas na construção da parte visual da aplicação.

3.2.1 *Front-end*

O *front-end* [3] é a aplicação que será executada no lado do cliente e ficará disponível no navegador. Nessa interface, o usuário pode interagir com os elementos visuais e disparar requisições para o lado do servidor (*back-end*) para visualizar ou manipular dados armazenados no banco de dados.

3.2.2 *Back-end*

O *back-end* [3] é a aplicação que é executada no lado do servidor, responsável por receber requisições enviadas pelo cliente (*front-end*) e acessar, ou processar, as informações contidas no banco de dados. Ele também é responsável por iniciar o processo de cruzamento de dados pela inteligência artificial, quando requisitado pelo usuário.

3.2.3 *Mock* de dados

Um *mock* [4] é um objeto ou função que simula o comportamento de partes externas do código de testes, como dependências, dados retornados pelo *back-end*, ou funções utilizadas por componentes, que não podem ser executadas no ambiente de testes.

3.3 Tecnologias utilizadas

Nesta seção, serão apresentadas algumas das tecnologias utilizadas no desenvolvimento do *front-end* da ferramenta e na solução proposta.

3.3.1 React.js

React [5] é uma biblioteca de JavaScript criada pelo Facebook, focada na construção de aplicações web com base em componentes. Cada componente é uma parte independente e reutilizável do sistema. A combinação destes componentes forma a interface final do usuário. O React utiliza o padrão de arquivo JavaScript XML (JSX), que unifica a linguagem de descrição HTML e a linguagem de programação JavaScript, tornando assim mais fácil a construção e manutenção dos componentes. Este nome é oriundo da sua natureza "reativa", que realiza atualizações somente nas partes da interface que foram alteradas, através de variáveis específicas denominadas de *states* (estados) [6].

[3]Disponível em: https://aws.amazon.com/pt/compare/the-difference-between-frontend-and-backend/. Acessado em 27 de outubro de 2024

[4]Disponível em: https://www.alura.com.br/artigos/testes-com-mocks-e-stubs. Acessado em 27 de outubro de 2024.

[5]Disponível em: https://pt-br.react.dev/. Acessado em: 25 de setembro de 2024

[6]Disponível em: https://pt-br.react.dev/learn/managing-state. Acesso em 24 de setembro de 2024.

3.3.2 Redux

Cada componente de uma aplicação construída em React possui seus próprios estados e atualiza de forma independente, conforme eles mudam. Para unificar os dados da interface, e disparar atualizações em componentes, mesmo sem ocorrer alterações em estados definidos dentro de seu escopo, é necessário utilizar um contexto de estados globais. Redux [7] é uma biblioteca de gerenciamento de estados globais da aplicação, que facilita o compartilhamento de informações entre partes do sistema. O Redux tem como principal funcionalidade centralizar estados, permitindo que qualquer componente da árvore de elementos possa acessar e alterar seus valores, sem que estejam declarados dentro do seu escopo ou sendo enviados de maneira direta.

3.3.3 TypeScript

TypeScript [8] é um subconjunto do JavaScript desenvolvido pela Microsoft, que adicona tipos estáticos ao código e permite também a criação de tipos personalizados, com o objetivo de melhorar a experiência de desenvolvimento de projetos. Sua principal função é ajudar a identificar erros de forma antecipada, durante o processo de desenvolvimento. Por ser uma extensão do JavaScript, o código TypeScript é compilado para JavaScript padrão, garantindo compatibilidade com todos os navegadores modernos. Ele é adicionado ao projeto visando facilitar a manutenção e análise de erros.

3.3.4 Jest

Jest [9] é um framework de testes para JavaScript, desenvolvido pelo Facebook, amplamente utilizado para testar projetos feitos em JavaScript e aplicações React. Ele oferece uma abordagem simples e intuitiva para escrever, documentar e executar rotinas. O Jest é conhecido por sua facilidade de configuração, além de possuir diversas ferramentas para facilitar criação de cenários fícticios, que representam bem as interações reais das funções do JavaScript.

3.3.5 Testing-library/react

Testing-library/react [10] é uma biblioteca de testes para React, com foco em testar componentes, de maneira mais próxima possível da forma como são utilizados pelos usuários finais. Ela promove testes baseadas na interação do usuário, simulando cliques

[7] Disponível em: https://react-redux.js.org/. Acessado em: 25 de setembro de 2024
[8] Diponível em: https://www.typescriptlang.org/docs/. Acessado em 24 de setembro de 2024
[9] Disponível em: https://jestjs.io/pt-BR/. Acessado em 24 de setembro de 2024.
[10] Diponível em: https://testing-library.com/. Acessado em: 24 de setembro de 2024.

com o mouse e outros eventos que podem ser feitos por ele, estimulando que a escrita dos testes verifique a funcionalidade e comportamento dos componentes em cenários reais. Essa abordagem contribui para criação de testes mais confiáveis e robustos e funciona em conjunto com a biblioteca Jest, mencionada anteriormente.

3.4 Classificação de Testes de Software

Este trabalho segue as classificações de teste de software conforme descritas por Glenford Myers, no livro *The Art of Software Testing* (Myers, 2012). Os conceitos definidos nesta obra são amplamente reconhecidos e adotados para escrever e estruturar testes de sitemas, separando-os em testes unitários e testes de integração. Esta classificação guiará a análise e os métodos aplicados na solução.

Os conceitos de testes unitários, integração e testes de aplicação descritos na obra de Myers são a base que levou ao conceito da pirâmide de testes, exibida na Figura 8, amplamente utilizados na modernidade. Esta pirâmide divide os testes em diferentes níveis, tendo na sua base os testes unitários (que validam o comportamento isolado do software), teste de integração (asseguram que as partes do sistema se comuniquem de maneira correta) e testes de aplicação ou *end-to-end*, que validam o fluxo seguido pelo usuário durante a utilização da ferramenta.

Figura 8: Pirâmide de Testes

4 METODOLOGIA

4.1 Visão Geral

O objetivo da implementação da rotina de testes automatizados é garantir que o software atende aos requisitos estabelecidos e apresenta o funcionamento esperado. Os testes foram desenvolvidos para a aplicação *front-end* do Hunter, verificando os componentes da tela "Explorar dados", visando assegurar o comportamento descrito no capítulo 2.

Ao todo, foram desenvolvidos 89 casos de teste, validando o estado dos componentes da interface após interações do usuário, a comunicação entre os componentes e o estado global da interface e a comunicação entre *front-end* e *back-end*. Por fim, utilizando as mesmas funções de testes que asseguram estes comportamentos, e utilizando ferramentas que simulam a interação do usuário, podemos verificar o fluxo de utilização da aplicação como um todo, tendo assim os testes de sistema e aceitação.

4.2 Classificação

Os testes podem ser classificados em testes unitários, testes de integração e testes de aceitação, e foram implementados utilizando as bibliotecas Testing-library/react e Jest, que juntas permitem criar scripts de verificação para componentes React, feitos em JS ou TS simulando o comportamento de um usuário acessando o sistema.

- **Testes Unitários**: verificam apenas uma parte independente do sistema, neste caso, um componente. Ele assegura que o componente executa o comportamento esperado para cada entrada de forma única, afim de evitar erros de interface. A biblioteca Testing-library/react exporta um objeto chamado *fireEvent*, que é capaz de disparar eventos da DOM [11], e foi utilizado para simular cliques e entradas de dados feitas pelo usuário.

- **Testes de Integração**: validam a comunicação entre as partes de um sistema, neste caso, a integração dos componentes com o estado global da aplicação. Eles verificam se os componentes respondem corretamente a alterações feitas nas variáveis globais, disparadas por outros componentes da interface. Eles também testam a comunicação entre a interface *front-end* e a aplicação *back-end*, garantindo que os dados estão sendo enviados da maneira correta.

- **Testes de Aceitação**: validam o funcionamento da aplicação como um todo, assegurando o comportamento correto através da simulação do fluxo de utilização

[11] DOM ou Document Object Model: interface de programação que representa a esturutura de documentos HTML e XML. Disponível em: https://www.w3.org/TR/DOM-Level-3-Core/

que um usuário faria ao acessar a ferramenta. Eles são implementados unificando as funções de testes unitários e de integração, validando a saída correta para as entradas e cliques feitos pelo usuário.

4.3 Configuração do Projeto

Inicialmente, precisamos definir os critérios de seleção de cenários para casos de testes. É fundamental que a rotina de testes cubra os aspectos descritos nos requisitos funcionais apresentados no cápitulo 2, assim como os requisitos não funcionais que garantem a comunicação correta entre as partes da ferramenta.

Os cenários de testes foram escolhidos com base no impacto que podem causar no funcionamento da ferramenta, visando evitar erros na inclusão de informações e envio de dados de maneira incorreta que acarretariam em problemas não mapeados pela equipe de desenvolvimento.

Antes de começar a desenvolver os casos de testes, é necessário adicionar algumas configurações no projeto, para que ele consiga executar os arquivos criados.

4.3.1 Dependências

A interface do Hunter é desenvolvida com React e TS. A etapa inicial para implementar a solução sugerida é instalar as seguintes bibliotecas como dependências do projeto:

1. **Testing-library/react**: Biblioteca para teste de componentes React;
2. **Testing-library/jest-dom**: Fornece um conjunto de *jest matchers* [12], possibilitando construir testes mais claros e fáceis de manter;
3. **Testing-library/user-event**: Tenta simular os eventos, conforme acontecem no ambiente de execução;
4. **Jest**: Biblioteca para testes de códigos JavaScript;
5. **Jest-environment-jsdom**: Simula um ambiente DOM igual ao navegador, garantindo que os testes representam o uso real;
6. **@types/jest**: Contém as definições dos tipos personalizados do Jest.

[12]No jest, utilizam-se "matchers"para verificar se um valor corresponde ao valor esperado, como no exempo *expect(valor).toBe(esperado)*, onde *toBe* verifica a igualdade entre os dois valores. Disponível em: https://jestjs.io/docs/using-matchers

4.3.2 Jest

Após instalação das dependências, é necessário adicionar um arquivo contendo as configurações para o Jest. Este arquivo precisa ser criado com o nome padrão *jest.config.cjs*, e define como a biblioteca deve se comportar ao executar a rotina de testes implementada. A seguir estão as principais configurações que podem ser definidas:

- ***testEnvironment***: Define o ambiente de testes que será utilizado;
- ***collectCoverage***: Permite que a biblioteca avalie a cobertura dos testes implementados;
- ***collectCoverageFrom***: Permite selecionar quais arquivos serão avaliados na cobertura dos testes;
- ***setupFilesAfterEnv***: Executa um arquivo de configuração adicional antes de cada teste;
- ***moduleNameMapper***: Mapeia importações de arquivos estáticos, como dependências e componentes.

A Figura 9 mostra como foi definido o arquivo de configurações do Jest para o Hunter 2.0.

```
/** @type {import("jest").Config} */
module.exports = {
  testEnvironment: 'jsdom',
  collectCoverage: true,
  collectCoverageFrom: ['src/**/*.tsx'],
  setupFilesAfterEnv: ['<rootDir>/src/setupTests.ts'],
  moduleNameMapper: {
    '^@/(.*)$': '<rootDir>/src/$1',
    '\\.(css|less|scss|sass)$': 'identity-obj-proxy',
    '^@components/(.*)$': '<rootDir>/src/components/$1',
    '^@lib/(.*)$': '<rootDir>/src/lib/$1',
    '^antd/es/(.*)$': 'antd/lib/$1',
    '^lodash-es/(.*)$': 'lodash/$1',
    '^lodash-es': 'lodash',
  },
  moduleDirectories: ['node_modules', 'src'],
  transform: {
    '^.+\\.tsx?$': 'babel-jest',
    '^.+\\.(js|jsx|mjs|cjs)$': 'babel-jest',
  },
  modulePathIgnorePatterns: ['<rootDir>/build/'],
};
```

Figura 9: Configurações do Jest.
Fonte: própria do autor

4.3.3 Testing-library/react

O Jest permite definir um arquivo para ser executado antes dos casos de testes. Logo, é uma boa prática deixar importações e constantes necessárias para a biblioteca Testing-library/react definidas neste arquivo, normalmente chamado de *setupTests*. Isto evita código duplicado e permite centralizar configurações que são necessárias para o ambiente de testes.

4.3.4 Execução

Por fim, para tornar possível a execução da rotina de testes, é necessário alterar o arquivo *package.json*, na origem do projeto da interface, que contém informações sobre o projeto, como suas dependências, scripts e versão. A alteração principal é a criação de um novo script, chamado "*test*" que usará o *framework* Jest para executar a rotina de testes.

4.4 Estrutura de um teste

Esta seção descreve como os testes da ferramenta foram construídos.

4.4.1 Organização

Os scripts foram desenvolvidos de maneira modular, colocando os testes de cada componente no seu diretório fonte, facilitando a manutenção e organização. Eles são organizados em blocos que descrevem o componente que estão testando e o comportamento que desejam assegurar. A Figura 10 mostra como é organizado um script de teste, que valida o componente *MainContent*.

```
1  import React from 'react';
2  import { api as mockApi } from '@lib/__mocks__/api';
3  import { hooks as mockHooks } from '@/lib/__mocks__/hooks';
4  import renderWithStore from '../test-utils';
5  import * as hooks from '@lib/hooks';
6
7  import ColumnList from './index';
8  import { fireEvent, waitFor } from '@testing-library/react';
9  import { initialState as mockInitialState } from '@/lib/__mocks__/initialState';
10 import './style.css';
11 import { RootState } from '@/lib/store';
12
13 jest.mock('@/lib/api', () => mockApi);
14       You, 3 months ago • testes
15 jest.mock('@lib/hooks/useColumns', () => mockHooks.useColumns);
16
17 describe('Main Content', () => {
18   test('Exibe selecionar todos', () => {
19     const { getByTestId } = renderWithStore(<ColumnList />, { initialState: mockInitialState });
20
21     expect(getByTestId('select-all')).toBeInTheDocument();
22   });
```

Figura 10: Arquivo de teste do *MainContent*
Fonte: própria do autor

Ao falhar em um script, o Jest exibe qual o caso de teste causou o erro, utilizando a estrutura descrita para direcionar o desenvolvedor para a seção correta do código que apresentou a falha, como exibido na Figura 11.

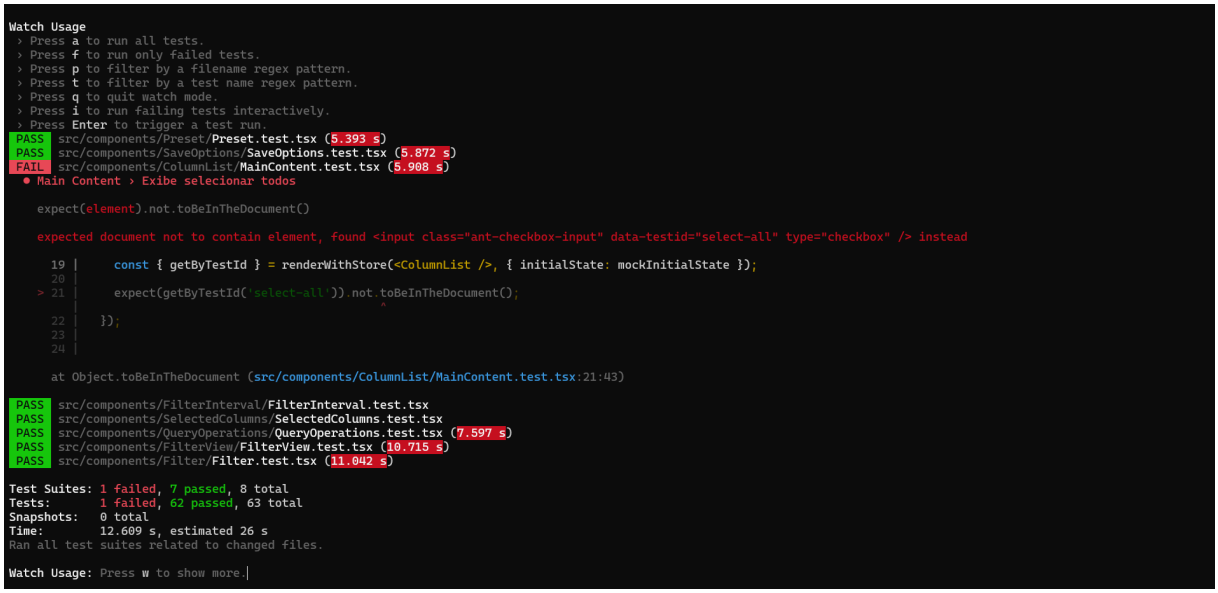


Figura 11: Erro exibido no terminal que executou o comando de teste
Fonte: própria do autor

4.4.2 Validação da saída

A biblioteca Testing-library/jest-dom fornece uma função chamada *expect*, ela recebe um valor como parâmetro e possui uma série de Jest *matchers*, que validam o parâmetro inserido. A seguir estão os principais *matchers* utilizados para desenvelvor a solução.

- ***toBeInTheDocument***: avalia se um elemento está presente na DOM;
- ***toHaveLength***: avalia o tamanho de uma lista de dados;
- ***toHaveBeenCalled***: avalia se uma função foi chamada;
- ***toHaveBeenCalledWith***: avalia se a função foi chamada, e os valores que foram enviados como atributos.

4.5 *Mock* de Dados

O ambiente de teste é diferente do ambiente de desenvolvimento, ele não executa a aplicação como um todo e normalmente não possui conexão com o *back-end*. Desta forma, torna-se necessário desenvolver uma forma de representar os dados que seriam exibidos durante a execução normal da ferramenta. Para isto, faz-se uso dos *mocks* de dados e funções, que são dados/funções fícticios pré-definidos, que possuem a mesma estrutura dos reais.

4.5.1 Banco de dados

Na solução implementada, foram criados *mocks* para as funções de busca de dados no *backend*, afim de validar os dados enviados para a outra aplicação. Isto é feito com o intuito de diminuir o tempo de execução dos testes, eliminando a necessidade de esperar o tempo necessário para comunicação entre *front-end* e *back-end* para envio dos dados existentes. Este trabalho visa validar somente o funcionamento da interface, e testes que garantem o tempo de execução e funcionamento da infraestrutura real serão implementados em trabalhos futuros.

A biblioteca Jest disponibiliza uma forma de criar *mocks* de funções que são exportadas de outros arquivos, assim, conseguimos substiuir as funções que são utilizadas no componente por uma função especialmente desenvolvida para o teste. A Figura 12 mostra como é criado o *mock* para a função que faz conexão com a API e busca a lista de campos diponíveis para seleção.

```
import mockDayJs from 'dayjs';
import { api } from './api';
import { CustomQuery, SavedQuery } from '../types';
import { customQueries as mockCustomQueries } from './custom-queries';

export const hooks = {
  useColumns: jest.fn().mockImplementation(() => {
    api.post('/description');

    return {
      columns: [
        {
          desc: 'Índice de fatonfe',
          id: 'id',
          name: 'id_fatonfe',
          desc2: 'Índice de Fatonfe',
          table: 'fatonfe',
          group: 'Todos',
          null: false,
        },
        {
          desc: 'Segunda',
          id: '2',
          name: 'second',
          desc2: 'Segunda coluna',
          table: 'fatoitem',
          group: 'Todos',
          null: false
        },
      ],
    };
  }),
  useDownloads: jest.fn().mockReturnValue({ downloads: [] }),
```

Figura 12: *Mock* da conexão entre a interface e o servidor da aplicação.
Fonte: própria do autor

4.5.2 Estado global

Também foi necessário criar dados para o estado global da aplicação, de forma que torne-se possível testar a integração dos componentes com o Redux, eliminando a necessidade de executar a aplicação inteira para testar uma seção específica. Eles foram desenvolvidos utilizando objetos estáticos, como pode ser visto na Figura 13.

Para ter acesso ao estado global *mockado*, precisamos fornecer um provedor desses dados para renderizar junto com o componente que será testado. Para isto, foi desenvolvido um *wrapper*, que envolve o componente de teste e fornece o contexto dos dados para serem utilizados por ele. Este *wrapper* pode ser usado com o estado inicial vazio, ou enviado uma versão customizada para testar um comportamento especifíco. Ele também retorna o estado global atual, permitindo que possamos fazer verificações dos valores de saída. A Figura 14 mostra como o *renderWithStore* foi implementado.

```
import { RootState } from '../store';

export const initialState: RootState = {
  selectedColumns: [],
  incorrectFilters: [],
  mainContent: {
    blockingFilters: [],
    fieldSearch: '',
    selectAll: false,
    queryMode: 'nfe'
  },
  columnFilters: { filters: [], key: 0, incorrectFilters: [] },
  layout: { showMessage: false, clearSearch: false, sidebarOpen: false },
  alertBox: {
    showAlert: false
  },
  groupFilter: '',
  showWarning: [],
  user: {
    groups: [],
    downloadName: '',
    downloadFormat: 'csv',
    downloadDelimiter: ';',
    columnName: 'extenso',
    stockVisibleColumns: [true, true, true, true, true, true, true, true, true, true, true, true],
    id: 'poc',
    currentGroup : 'anon'
  }
};      You, 2 months ago • testes com mock da api
```

Figura 13: *Mock* do estado global inicial da interface.
Fonte: própria do autor

```tsx
import React, { PropsWithChildren } from 'react';
import { RenderOptions, render as rtlRender } from '@testing-library/react';
import { EnhancedStore, configureStore } from '@reduxjs/toolkit';
import { Provider } from 'react-redux';
import { RootState, reducer } from '@lib/store';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

type RenderWithStoreProps = {
  initialState?: RootState;
  store?: EnhancedStore;
} & Omit<RenderOptions, 'queries'>;

const queryClient = new QueryClient();

function renderWithStore(
  ui: React.ReactElement,
  {
    initialState,
    store = configureStore({
      reducer,
      preloadedState: initialState,
    }),
    ...renderOptions
  }: RenderWithStoreProps = {}
) {
  function Wrapper({ children }: PropsWithChildren) {
    return (
      <QueryClientProvider client={queryClient}>
        <Provider store={store}>{children}</Provider>
      </QueryClientProvider>
    );
  }

  return {
    ...rtlRender(ui, { wrapper: Wrapper, ...renderOptions }),
    store,
  };
}

export default renderWithStore;      You, 3 months ago • render with providers
```

Figura 14: *Wrapper* que fornece um estado global.
Fonte: própria do autor

4.6 Testes Unitários

Nesta seção, serão exibidos alguns exemplos de testes unitários que foram implementados.

4.6.1 Conteúdo Principal

O teste mostrado na Figura 15, verifica se o conteúdo principal exibe a lista de itens disponíveis para consulta.

```
test('Renderiza as colunas', () => {
  const { getByText } = renderWithStore(<ColumnList />, {});

  expect(getByText('Índice de fatonfe')).toBeInTheDocument();
  expect(getByText('Segunda')).toBeInTheDocument();
});
```

Figura 15: Garante exibição dos nomes das colunas disponíveis.
Fonte: própria do autor

4.6.2 Filtro

A Figura 16 contém um teste unitário que assegura que o componente de filtro exibe os campos de "Início" e "Fim" para filtros com a opção "intervalo" marcada.

```
test('Exibe campos para início e fim', () => {
  const { getByText } = renderWithStore(
    <FilterInterval
      editable
      filter={initialState.columnFilters.filters[0]}
      maxValueField="maxValue"
      minValueField="minValue"
      weekValueField="week"
    />,
    { initialState }
  );

  expect(getByText('Início:')).toBeInTheDocument();
  expect(getByText('Fim:')).toBeInTheDocument();
});
```

Figura 16: Verifica os valores do filtro com a opção "intervalo" selecionada.
Fonte: própria do autor

4.7 Testes de Integração

A seguir temos alguns exemplos de como são implementados os teste de integração entre módulos da ferramenta.

4.7.1 Redux

O testes de integração com o redux são feitos avaliando o estado que é retornado pelo *renderWithStore*, após interação do usuário com o sistema. A Figura 17 mostra um teste que verifica se o botão "limpar colunas" remove todas as colunas selecionadas para busca. O teste seleciona o botão através de seu id de testes e avalia o estado global retornado, após clicar no botão.

```
test('Remove as colunas selecionadas', () => {
  const { getByText, getByTestId, store } = renderWithStore(<SelectedColumns />, { initialState });
  expect(getByText('Índice de fatonfe')).toBeInTheDocument();

  const removeBtn = getByTestId('remove-all-cols');
  fireEvent.click(removeBtn);
  expect(getByText('Nenhuma coluna selecionada')).toBeInTheDocument();
  expect(store.getState().selectedColumns).toHaveLength(0);
});
```

Figura 17: Assegura que a interface remove colunas selecionadas.
Fonte: própria do autor

4.7.2 *Back-end*

Os testes de integração entre *front-end* e *back-end* avaliam apenas os objetos enviados pela ferramenta, e os *endpoints* que estão sendo acessados. A Figura 18 mostra o caso de teste que valida a ação de pré-visualização de dados, disponível através do botão "Visualizar" na base da coluna Filtros. Esta ação remove os filtros que não estão preenchidos, visando evitar erros na busca, e envia os dados selecionados para o *endpoint* */filter*.

```
  test('Remove filtros vazios antes de enviar a requisição', async () => {
    const filledFilter = {...fatonfeFilter, value: '111'};

    initialState = {
      ...initialState,
      selectedColumns: ['id'],
      columnFilters: {
        ...initialState.columnFilters,
        filters: [
          filledFilter,
          fatoItemNFeFilter
        ]
      }
    };
    const { getByText } = renderWithStore(<QueryOperations />, { initialState });

    const sendButton = getByText('Visualizar');
    expect(sendButton).not.toBeDisabled();
    fireEvent.click(sendButton);

    await waitFor(() => {
      expect(mockApi.post).toHaveBeenCalledWith(
        'filter',
        expect.objectContaining({
          mode: 'preview',
          limit: 100,
          columns: expect.objectContaining({
            fatonfe: ['id_fatonfe'],
          }),
          filters: {
            fatonfe: [filledFilter]
          }
        })
      );
    });
  });
});
```

Figura 18: Verifica os dados enviados pela interface na opção “Visualizar”.
Fonte: própria do autor

5 APRESENTAÇÃO DOS RESULTADOS

Neste capítulo, serão apresentados os resultados obtidos, e como validamos a solução descrita no capítulo anterior.

5.1 Testes desenvolvidos

Esta seção visa apresentar os principais testes desenvolvidos para assegurar o requisitos funcionais mencionados no capítulo 2, separados por seções da interface e categoria de testes.

Tabela 1: Testes Conteúdo Principal

Tipo de Teste	Nome da função	Descrição
Unitário	*showSelectAllColumns*	Verifica se o *checkbox* que seleciona todas as colunas está visível
Unitário	*displayColumns*	Verifica se as colunas disponíveis estão sendo exibidas
Unitário	*displayColumnsDesc-BtnOnHover*	Verifica se o botão que exibe a descrição da coluna fica visível ao posicionar o mouse sobre o componente
Unitário	*displayColumnDescription*	Verifica se a descrição da coluna é exibida ao clicar no ícone de informações
Unitário	*displayColumnList-ErrorMessage*	Verifica se a mensagem de erro é exibida caso o *backend* não retorne os dados de colunas
Unitário	*displayLoading-SkeletonColumnList*	Verifica se a ferramenta exibe a animação de carregamento de dados enquanto a requisição está sendo executada
Integração	*makeColumnsListRequest*	Verifica se a ferramenta faz a requisição das colunas disponíveis para consulta
Integração	*selectAllColumns*	Verifica se todas as colunas são enviadas para o estado global ao clicar no *checkbox* de selecionar todas

Integração	*selectOneColumn*	Verifica se as colunas selecionadas individualmente são enviadas para o estado global
Integração	*removeOneSelectedColumn*	Verifica se as colunas são removidas do estado global ao clicar em uma coluna que já está selecionada
Integração	*selectOneFilter*	Verifica se os filtros selecionados são enviados para o estado global

Tabela 2: Testes dos Filtros

Tipo de Teste	Nome da função	Descrição
Unitário	*filterShowCorrectInput*	Verifica se o componente de filtro exibe o *input* correto com base no tipo de dado da coluna
Unitário	*filterBorderRed*	Verifica se o componente de filtro destaca o campo que possui erro de preenchimento
Unitário	*filterBottomDisplay-IntervalCheckbox*	Verifica se o componente de filtro mostra a opção Intervalo para filtros númericos e de data
Unitário	*filterBottomDisplay-OrOption*	Verifica se o filtro exibe a opção Ou corretamente
Unitário	*filterBottomDisplay-NullCheckbox*	Verifica se o filtro exibe a opção Null para filtros que podem ser preenchidos com este dado
Unitário	*filterBottomDisplay-NotCheckbox*	Verifica se o filtro exibe a opção Não
Unitário	*filterDisplaySecondInput*	Verifica se o filtro exibe o segundo campo de preenchimento quando a opção Ou está ativada
Unitário	*filterDisplayIntervalFields*	Verifica se o filtro exibe os campos de início e fim quando a opção Intervalo está marcada

Unitário	*filterDisplayIntervalFields*	Verifica se o filtro exibe os campos de início e fim para a segunda opção quando a opção Ou e Intervalo 2 está marcada
Unitário	*filterDisplayIntervalFields*	Verifica se o filtro exibe os campos de início e fim para para os dois valores que serão inseridos quando a opção Intervalo 1 e Intervalo 2 e Ou estão marcados
Unitário	*filterDisplayErrorMessage*	Verifica se o filtro exibe a mensagem de erro referente ao valor inserido no campo principal
Unitário	*filterDisplayErrorMessage*	Verifica se o filtro exibe a mensagem de erro referente ao valor inserido no segundo campo com a opção Ou marcada
Unitário	*filterDisplayErrorMessage*	Verifica se o filtro realiza a validação do número de CNPJ inserido no filtro e exibe mensagem de erro caso esteja incorreto
Unitário	*filterDisplayErrorMessage*	Verifica se o filtro realiza corretamente validação de múltipos números de CNPJ inseridos no filtro exibindo a mensagem de erro somente para os valores incorretos
Unitário	*filterValidateInterval*	Verifica se o filtro valida os valores inseridos no intervalo principal
Unitário	*filterValidateInterval*	Verifica se o filtro valida os valores inseridos no intervalo secundário com a opção Ou ativada
Unitário	*displayFilterDescription-OnHeader*	Verifica se o filtro exibe a descrição da coluna no cabeçalho
Unitário	*filterChangeExibtionMode*	Verifica se o filtro altera o modo de exibição entre expandido e resumido

Integração	*filterChangeIntervalValue*	Verifica se a informação do intervalo do filtro é enviada para o estado global ao clicar no *checkbox*
Integração	*filterChangeOrValue*	Verifica se a informação de Ou do filtro é enviada para o estado global ao clicar no *checkbox*
Integração	*filterChangeInterval-TwoValue*	Verifica se a informação do segundo intervalo do filtro é enviada para o estado global ao clicar no *checkbox*
Integração	*filterChangeNotValue*	Verifica se a informação do *checkbox* Não do filtro é enviada para o estado global
Integração	*filterChangeMainValue*	Verifica se o valor inserido no campo principal do filtro é enviado para o estado global
Integração	*filterChangeValueTwo*	Verifica se o valor inserido no segundo campo do filtro é enviado para o estado global
Integração	*removeFilter*	Verifica se o filtro é removido do estado global ao clicar no botão para remover

Tabela 3: Testes das Requisições

Tipo de Teste	Nome da função	Descrição
Unitário	*disableDownloadButton*	Verifica se o botão de *download* está desativado quando não há um filtro de data ou de chave de acesso
Unitário	*disablePreviewButton*	Verifica se o botão de visualizar está desativado quando não há uma coluna selecionada

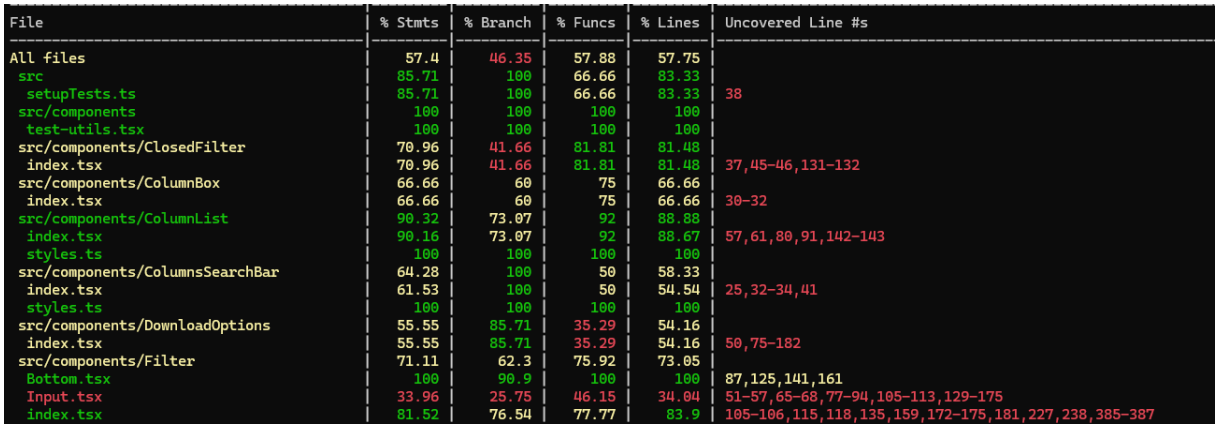
Tipo de Teste	Nome da função	Descrição
Unitário	*enableDownloadButton*	Verifica se o botão de *download* está ativado quando há um filtro de data ou de chave de acesso
Unitário	*enablePreviewButton*	Verifica se o botão de visualizar está ativado quando há pelo menos uma coluna selecionada
Unitário	*enableSaveQueryButton*	Verifica se o botão de salvar consulta está ativado quando há pelo menos uma coluna selecionada
Unitário	*disableSaveQueryButton*	Verifica se o botão de salvar consulta está desatiovado quando não há uma coluna selecionada
Integração	*sendDataOnRequest*	Verifica se o botão de visualizar envia as colunas e filtros preenchidos para o *backend* com o parâmetro preview e limite de 100 dados retornados
Integração	*sendDataOnRequest*	Verifica se o botão de *download* envia as colunas e filtros preenchidos para o *backend* com o parâmetro download e sem limite de dados retornados
Integração	*removeEmptyData*	Verifica se o botão de visualizar remove os filtros que não estão preenchidos da requisição
Integração	*formatNCMfilters*	Verifica se o botão de visualizar formata os filtros de NCM inserindo a porcentagem para valores com menos que 8 digitos
Integração	*saveCustomQuery*	Verifica se o botão de salvar consulta envia os dados selecionados pelo o usuário para o *backend*

Tabela 4: Testes do Menu Lateral

Tipo de Teste	Nome da função	Descrição
Unitário	*showPresetItems*	Verifica se o menu lateral exibe os itens pré-configurados
Unitário	*showCustomItems*	Verifica se o menu lateral exibe as consultas criadas pelo usuário
Unitário	*displayRemoveButton*	Verifica se o menu lateral exibe o botão para remover consultas customizadas
Integração	*requestCustomQueries*	Verifica se a interface faz requisição pelas consultas configuradas pelo usuário
Integração	*selectCustomQueries*	Verifica se a interface envia os dados da consulta customizada ao ser selcionada pelo usuário
Integração	*removeCustomQueries*	Verifica se a interface faz requisição para remover consulta customizada enviado o id correto

5.2 Coberturas dos testes desenvolvidos

A biblioteca Jest fornece um relatório detalhado, ao final da execução dos scripts de teste, que mostra o quanto está sendo coberto pelos casos de testes implementados. A Figura 19 mostra parte da tabela com a cobertura dos testes que foram implementados, a primeira coluna mostra a lista de arquivos que estão sendo testados, a segunda mostra a quantidade de declarações do código que estão validadas, na terceira é exibida a quantidade de ramificações (*if*, *else*, *switch*) que estão sendo cobertas, a quarta e quinta coluna mostram a quantidade de funções e linhas de códigos executadas durantes os testes e a última coluna mostra um sumário de linhas que ainda precisam ser testadas.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	57.4	46.35	57.88	57.75	
src	85.71	100	66.66	83.33	
setupTests.ts	85.71	100	66.66	83.33	38
src/components	100	100	100	100	
test-utils.tsx	100	100	100	100	
src/components/ClosedFilter	70.96	41.66	81.81	81.48	
index.tsx	70.96	41.66	81.81	81.48	37,45-46,131-132
src/components/ColumnBox	66.66	60	75	66.66	
index.tsx	66.66	60	75	66.66	30-32
src/components/ColumnList	90.32	73.07	92	88.88	
index.tsx	90.16	73.07	92	88.67	57,61,80,91,142-143
styles.ts	100	100	100	100	
src/components/ColumnsSearchBar	64.28	100	50	58.33	
index.tsx	61.53	100	50	54.54	25,32-34,41
styles.ts	100	100	100	100	
src/components/DownloadOptions	55.55	85.71	35.29	54.16	
index.tsx	55.55	85.71	35.29	54.16	50,75-182
src/components/Filter	71.11	62.3	75.92	73.05	
Bottom.tsx	100	90.9	100	100	87,125,141,161
Input.tsx	33.96	25.75	46.15	34.04	51-57,65-68,77-94,105-113,129-175
index.tsx	81.52	76.54	77.77	83.9	105-106,115,118,135,159,172-175,181,227,238,385-387

Figura 19: Cobertura dos casos de teste
Fonte: própria do autor

5.3 Testes de aceitação

Os testes de aceitação são os casos de teste que validam o fluxo de utilização da ferramenta pelo auditor fiscal. Eles foram implementados reutilizando as funções de testes unitários e de integração descritas anteriormente, agrupando-as em um cenário mais amplo. Desta forma, conseguimos assegurar o funcionamento correto da ferramenta e os fluxos alternativos e de exceção.

As seções a seguir descrevem como foram validados os principais requistos mencionados anteriormente.

5.3.1 Usuário pode fazer consultas personalizadas

Para criar este cenário foram utilizadas as funções: *makeColumnsListRequest*, *displayColumns*, *selectOneColumn*, *selectOneFilter*, *filterShowCorrectInput*, *filterChangeMainValue*, *filterDisplayErrorMessage*, *enablePreviewButton*, *sendDataOnRequest*.

5.3.2 Usuário pode fazer *downloads* personalizados

Este cenário foi validado com as seguintes funções: *makeColumnsListRequest*, *displayColumns*, *selectOneColumn*, *selectOneFilter*, *enableDownloadButton*, *filterShowCorrectInput*, *filterChangeMainValue*, *filterDisplayErrorMessage*, *sendDataOnRequest*.

5.3.3 Usuário pode criar consultas customizadas

Para assegurar este comportamento, utilizamos as mesmas funções dos cenários anteriores, porém assegurando o envio para o *endpoint* correto com as seguintes funções: *makeColumnsListRequest*, *displayColumns*, *selectOneColumn*, *selectOneFilter*, *filterShowCorrectInput*, *filterChangeMainValue*, *filterDisplayErrorMessage*, *saveCustomQuery*.

5.3.4 Usuário pode utilizar consultas pré-configuradas

Para assegurar este requisito, precisamos verificar se a ferramenta executas as seguintes etapas: *showPresetItems*, *selectCustomQueries*, *sendDataOnRequest*

5.3.5 Usuário pode utilizar consultas customizadas

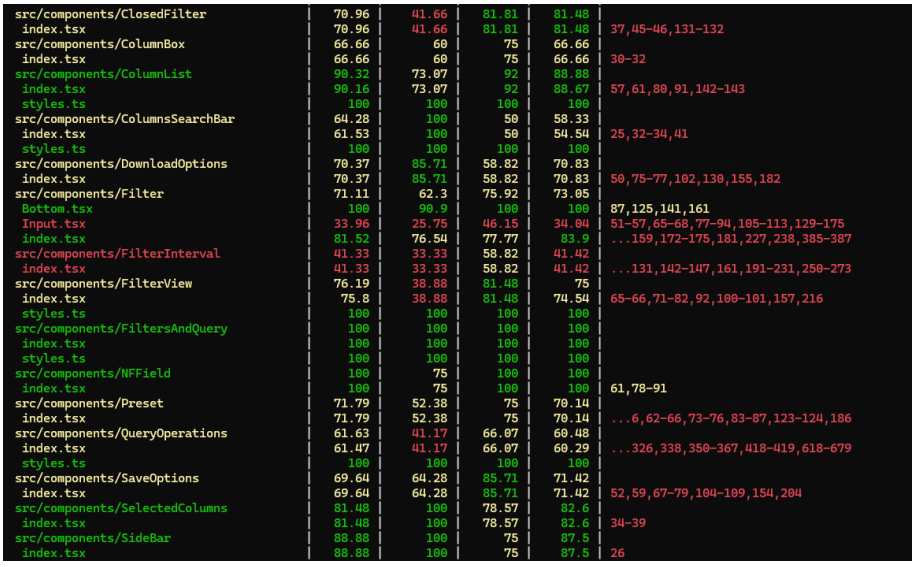
Este cenário é bem parecido com o fluxo anterior, apenas com algumas funções extras: *requestCustomQueries*, *showCustomItems*, *selectCustomQueries*, *selectOneFilter*, *selectOneColumn*, *filterShowCorrectInput*, *filterChangeMainValue*, *sendDataOnRequest*.

5.3.6 Usuário pode remover consultas customizadas

Neste cenário, precisamos garantir a busca pelas consultas customizadas e assegurar o envio do identificador único da consulta para o *endpoint* correto. *requestCustomQueries*, *showCustomItems*, *displayRemoveButton*, *removeCustomQueries*

5.4 Cobertura de testes Explorar Dados

A Figura 20 exibe a tabela de cobertura dos testes desenvolvidos para os componentes que fazem parte da tela "Explorar Dados"e nos fornece uma ideia geral do quanto das funcionalidades da interface estão validadas atualmente.



src/components/ClosedFilter	70.96	41.66	81.81	81.48	
index.tsx	70.96	41.66	81.81	81.48	37,45-46,131-132
src/components/ColumnBox	66.66	60	75	66.66	
index.tsx	66.66	60	75	66.66	30-32
src/components/ColumnList	90.32	73.07	92	88.88	
index.tsx	90.16	73.07	92	88.67	57,61,80,91,142-143
styles.ts	100	100	100	100	
src/components/ColumnsSearchBar	64.28	100	50	58.33	
index.tsx	61.53	100	50	54.54	25,32-34,41
styles.ts	100	100	100	100	
src/components/DownloadOptions	70.37	85.71	58.82	70.83	
index.tsx	70.37	85.71	58.82	70.83	50,75-77,102,130,155,182
src/components/Filter	71.11	62.3	75.92	73.05	
Bottom.tsx	100	90.9	100	100	87,125,141,161
Input.tsx	33.96	25.75	46.15	34.04	51-57,65-68,77-94,105-113,129-175
index.tsx	81.52	76.54	77.77	83.9	...159,172-175,181,227,238,385-387
src/components/FilterInterval	41.33	33.33	58.82	41.42	
index.tsx	41.33	33.33	58.82	41.42	...131,142-147,161,191-231,250-273
src/components/FilterView	76.19	38.88	81.48	75	
index.tsx	75.8	38.88	81.48	74.54	65-66,71-82,92,100-101,157,216
styles.ts	100	100	100	100	
src/components/FiltersAndQuery	100	100	100	100	
index.tsx	100	100	100	100	
styles.ts	100	100	100	100	
src/components/NFField	100	75	100	100	
index.tsx	100	75	100	100	61,78-91
src/components/Preset	71.79	52.38	75	70.14	
index.tsx	71.79	52.38	75	70.14	...6,62-66,73-76,83-87,123-124,186
src/components/QueryOperations	61.63	41.17	66.07	60.48	
index.tsx	61.47	41.17	66.07	60.29	...326,338,350-367,418-419,618-679
styles.ts	100	100	100	100	
src/components/SaveOptions	69.64	64.28	85.71	71.42	
index.tsx	69.64	64.28	85.71	71.42	52,59,67-79,104-109,154,204
src/components/SelectedColumns	81.48	100	78.57	82.6	
index.tsx	81.48	100	78.57	82.6	34-39
src/components/SideBar	88.88	100	75	87.5	
index.tsx	88.88	100	75	87.5	26

Figura 20: Cobertura de testes Explorar Dados
Fonte: própria do autor

6 CONCLUSÕES E TRABALHOS FUTUROS

A rotina de casos de testes se mostrou uma solução viável e eficiente para validar o comportamento da aplicação durante casos reais de uso. Pequenas alterações na implementação do projeto podem resultar em grandes falhas, as vezes difíceis de serem encontradas. Entretanto, a solução garante o funcionamento de todas as partes da interface individualmente e a sua comunicação, facilitando o controle de qualidade do produto final.

A primeira sugestão para trabalhos futuros, é manter a aplicação de boas práticas de desenvolvimento em ambiente ágil. Técnicas como as descritas em *Código limpo* (2008), enfatizam que variáveis e funções devem conter nomes claros e descritivos que ajudam a entender o cenário que o trecho de código é utilizado, facilitando assim a criação de testes mais precisos e que validam todos os fluxos presentes. Estas técnicas também impactam diretamente na qualidade dos casos de teste, permitindo a refatoração e melhoria das funcionalidades presentes, mantendo assim o código sempre atualizado, diminuindo riscos e removendo ambiguidades presentes no projeto que criariam a necessidade de mais casos de teste.

Os príncipios SOLID, mencionados em *Arquitetura limpa* (2019), devem ser sempre analisados e implementados quando possível. Dentre eles, vale mencionar o SRP (Príncipio da Responsabilidade Única) que enfatiza que funções e componentes da interface devem sempre ter um único objetivo, facilitando novamente a criação de testes e garantindo que todos os fluxos sejam abordados corretamente, aumentando também a componentização e organização do código, que tem um impacto direto na facilidade de manutenção da ferramenta. Outro príncipio bastante utilizado foi o DIP (Príncipio da Inversão de Depêndencias), que visa melhorar a arquitetura do software em geral, removendo funções de mais alto nível de componentes que precisam de algumas informações, como por exemplo consumo de dados do *back-end*, tornando mais simples a criação dos cenários de testes que precisam desta implementação. Na ferramenta analisada, este príncipio foi implementado de maneira indireta, utilizando os *hooks*[13] customizados que fazem consumo e proveem os dados necessários para componentes da interface, tornando mais fácil a criação de *mocks* de dados e de funções necessárias para o ambiente de testes, promovendo assim a reusabilidade das funções de validação escritas.

Devido ao tamanho da aplicação, é necessário continuar desenvolvendo novos casos de testes, que englobam outros comportamentos que não foram validados com a solução implementada e também asseguram o funcionamento das outras telas disponíveis na aplicação

Outra sugestão para trabalhos futuros é a criação de etapas de validação do teste,

[13]Disponível em: https://react.dev/learn#using-hooks. Acessado em: 17 de novembro de 2024.

como por exemplo executar a rotina de testes durante a execução do ambiente de desenvolvimento. Esta abordagem visa testar a interface em tempo real (utilizando outras dependências como React Concurrent Mode [14]), durante o processo de desenvolvimento, executando os testes a cada alteração feita por desenvolvedores. Uma outra solução seria conectar a rotina de testes com as alterações enviadas para ambiente de desenvolvimento (por meio de técnicas de integração contínua, utilizando ferramentas como GitHub Actions [15]), validando os componentes da interface após o desenvolvimento e aprovação de novas *features* (funcionalidades) para a aplicação.

Por fim, para garantir o funcionamento completo, recomenda-se o desenvolvimento de uma rotina de testes voltada para a outra parte da ferramenta, o *back-end*. Esta nova rotina teria como objetivo validar os métodos implementados no projeto do servidor, e a sua comunicação com o banco de dados e o *frontend*.

[14]Permite a execução simultânea de várias tarefas, possibilitando a execução da ferramenta e da rotina de testes. Disponível em: https://dev.to/codesensei/the-ultimate-guide-to-react-conquering-concurrent-mode-and-suspense-3ahb. Acesso em 24 de setembro de 2024.

[15]Ferramenta que permite criação e personalização de workflows, possibilitando a automação da execução dos testes a partir de uma alteração enviada. Disponível em: https://docs.github.com/pt/actions. Acesso em: 24 de setembro de 2024

REFERÊNCIAS

[1] MYERS, G. J. ***The Art of Software Testing***. 3. ed. Hoboken: John Wiley & Sons, 2012.

[2] FACEBOOK. *React*. Disponível em: https://pt-br.react.dev/. Acesso em: 24 de set. de 2024.

[3] MICROSOFT. *TypeScript*. Disponível em: https://www.typescriptlang.org/pt/docs/. Acesso em: 24 de set. de 2024.

[4] TESTING LIBRARY. *React Testing Library*. Disponível em: https://testing-library.com/. Acesso em: 24 de set. de 2024.

[5] REACT REDUX. *React-Redux*. Disponível em: https://redux.js.org/. Acesso em: 24 de set. de 2024.

[6] FACEBOOK. *Jest*. Disponível em: https://jestjs.io/pt-BR/. Acesso em: 24 de set. de 2024.

[7] DODDS, K. C. (2018). *Testing JavaScript Applications*. Disponível em: https://www.testingjavascript.com/. Acesso em: 24 de set. de 2024.

[8] WIERUCH, R. ***Code Organization and Testing***. In: Robin, W. . *The Road to Learn React* (1. ed.). Code with Ryan, 2019.

[9] MARTIN, R. C. ***Clean Code: A Handbook of Agile Software Craftsmanship*** . 1. ed. Upper Saddle River: Prentice Hall, 2008.

[10] MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design** . 1. ed. São Paulo: Alta Books, 2019.