

Introdução

Este documento descreve em detalhes a estruturação, configuração e execução dos testes unitários realizados no projeto API-2025.1-Frontend, desenvolvido com React e TypeScript. A escolha por implementar testes unitários tem como objetivo garantir a qualidade, a confiabilidade e a manutenção do código, assegurando que componentes funcionem conforme esperado e prevenindo regressões durante evoluções futuras do sistema. Foram adotadas ferramentas amplamente reconhecidas pela comunidade de desenvolvimento, como o Jest para execução dos testes e a React Testing Library (RTL) para facilitar a interação com os componentes de forma similar à experiência real do usuário.

Testes unitários são procedimentos essenciais para verificar o comportamento esperado de unidades isoladas do sistema, neste caso, componentes React, testando sua renderização e interações de forma independente das demais partes do sistema. Ao garantir que cada componente cumpre sua função, promove-se maior segurança na manutenção e evolução da aplicação.

Justificativa Técnica e Estratégia de Escolha

A escolha do Jest se deve à sua robustez, velocidade e integração facilitada com projetos em TypeScript, principalmente através do pacote ts-jest, que permite executar e compilar testes escritos em TypeScript sem a necessidade de configurações adicionais complexas. Além disso, o Jest oferece suporte nativo a mocks, testes assíncronos, relatórios de cobertura e execução paralela, características que favorecem a escalabilidade e a confiabilidade das suítes de teste (JEST, 2024).

Optou-se pela React Testing Library por sua filosofia centrada no comportamento do usuário, que privilegia a escrita de testes simulando interações reais ao invés de testar implementações internas (DOM-TESTING-LIBRARY, 2024). Isso reduz a fragilidade dos testes e melhora sua manutenção, pois alterações internas que não impactam o comportamento percebido pelo usuário não quebram os testes. O TypeScript foi escolhido para garantir segurança de tipos durante a escrita dos componentes e dos testes, detectando erros em tempo de desenvolvimento e tornando mais claras as interfaces (props) de cada componente.

Adotou-se também o padrão AAA (*Arrange, Act, Assert*) para a estruturação dos testes, organizando o código em três etapas: preparação do cenário, execução da ação e

verificação do resultado. Essa estrutura metodológica facilita a escrita, leitura e manutenção dos testes (MESZAROS, 2007).

Estrutura e Organização dos Arquivos de Teste

Os arquivos de teste foram organizados na pasta `tests/unit/components/ui/`, espelhando diretamente a estrutura dos componentes localizados em `src/components/ui/`. Essa correspondência direta facilita a manutenção, expansão e localização dos testes conforme novas funcionalidades são implementadas. Por exemplo, o componente `Applcon` possui seu teste correspondente no arquivo `Applcon.test.tsx`. Esse padrão foi aplicado de maneira uniforme para todos os componentes testados, como `Searchbar`, `AlertTypeForm` e `TableExemple`. A nomeação dos arquivos segue a convenção `NomeDoComponente.test.tsx`, permitindo que o Jest identifique automaticamente os arquivos de teste.

Configuração Técnica

Diversas dependências de desenvolvimento foram instaladas para garantir uma configuração adequada do ambiente de testes. Entre elas, destacam-se o Jest, o `ts-jest` para integração com TypeScript, e os tipos necessários via `@types/jest`. Além disso, foram incluídas as bibliotecas `@testing-library/react` e `@testing-library/jest-dom`, que adicionam funcionalidades específicas para a execução e validação dos testes.

Considerando que o projeto inclui componentes que importam arquivos `.css`, foi necessário configurar o Jest para evitar erros durante os testes. Para isso, foi adicionado ao `jest.config.ts` o `moduleNameMapper` com o valor `identity-obj-proxy`, permitindo que o Jest trate essas importações como mocks dinâmicos, evitando falhas comuns como `SyntaxError: Unexpected token '.'` (IDENTITY-OBJ-PROXY, 2024).

O arquivo `jest.config.ts` foi configurado com o preset como `ts-jest`, o `testEnvironment` como `jsdom` — essencial para simular o ambiente de um navegador — e o `setupFilesAfterEnv` para incluir extensões de matchers como `toBeInTheDocument()` (JEST, 2024). A configuração `esModuleInterop` foi ativada no `tsconfig.json` para garantir interoperabilidade entre módulos CommonJS e ESM, especialmente importante para importações padrão de bibliotecas como o React.

Escrita e Execução dos Testes

Todos os testes seguem a estrutura padrão AAA. Por exemplo, ao testar o componente `AppIcon`, o teste prepara o cenário renderizando o componente com um nome de ícone, executa a ação de renderização e verifica se o ícone está presente na árvore do DOM renderizada ou, caso o ícone não exista, se a função `console.warn` foi chamada corretamente. Isso assegura não apenas a funcionalidade principal, mas também o tratamento adequado de casos de erro.

Para componentes como `TableExemple`, que utiliza o `DataGrid` da MUI, verificou-se a presença do elemento grid na renderização, validando a semântica e a acessibilidade do componente. Todos os testes foram escritos utilizando tipagens estritas, proporcionadas pelo TypeScript, garantindo que erros comuns de tipo fossem detectados ainda em tempo de desenvolvimento.

A execução dos testes ocorre com o comando `npm test`, que invoca o Jest e executa automaticamente todas as suítes de teste identificadas. O Jest fornece uma saída detalhada no terminal, informando, para cada suíte, se foi aprovada (PASS) ou falhou (FAIL), o tempo de execução e detalhes sobre qualquer erro encontrado.

Como Rodar os Testes

A execução dos testes é realizada utilizando o comando `npm test`. Este comando invoca o Jest, que por sua vez localiza automaticamente todos os arquivos de teste com a extensão `.test.ts` ou `.test.tsx` dentro do projeto. Para que o ambiente esteja devidamente configurado, é imprescindível que todas as dependências de desenvolvimento estejam previamente instaladas, incluindo `jest`, `@types/jest`, `ts-jest`, `@testing-library/react` e `@testing-library/jest-dom`.

Durante a execução, o Jest fornece uma saída detalhada no terminal, informando, para cada suíte de testes, se ela foi aprovada (PASS) ou falhou (FAIL). Cada teste executado também apresenta o tempo de execução e, em caso de falha, o Jest exibe o erro ocorrido, a linha exata do código onde o erro foi detectado e uma breve descrição do comportamento divergente encontrado. O ambiente configurado também exibe avisos e depreciações no terminal, como as relacionadas ao pacote `punycode`, que, embora não afetem a execução dos testes, indicam dependências que podem ser futuramente atualizadas.

Pré-requisitos

Para rodar os testes, é necessário ter o Node.js previamente instalado, bem como todas as dependências do projeto, obtidas via npm install.

A execução padrão dos testes é realizada com o comando:

Comando padrão:

```
npm test
```

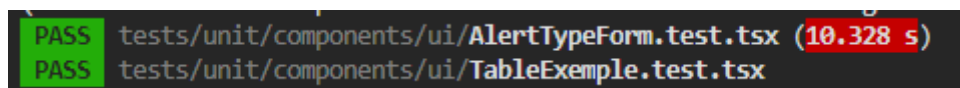
Ou com mais detalhes:

```
npx jest --verbose
```

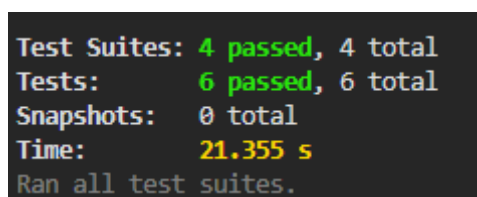
Durante a execução, o Jest localiza automaticamente todos os arquivos com extensão .test.ts ou .test.tsx. O terminal exibirá, para cada suíte de testes, o resultado (PASS ou FAIL), seguido de informações complementares como o tempo de execução e eventuais erros detectados.

O ambiente também exibe avisos e mensagens de depreciação, como as relacionadas ao pacote punycode, que embora não afetem a execução dos testes, indicam dependências que podem ser atualizadas futuramente.

Interpretação dos Resultados



```
PASS tests/unit/components/ui/AlertTypeForm.test.tsx (10.328 s)
PASS tests/unit/components/ui/TableExample.test.tsx
```



```
Test Suites: 4 passed, 4 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        21.355 s
Ran all test suites.
```

Ao final da execução do comando npm test, o Jest apresenta um resumo no terminal, indicando a quantidade de suítes de testes executadas, quantas foram bem-sucedidas e quantas falharam, bem como o tempo total de execução.

Significados principais da saída:

- PASS: indica que a suíte de testes foi aprovada.
- FAIL: indica que houve falha em alguma parte do teste.

- Test Suites: quantidade total de arquivos de teste.
- Tests: total de casos de teste executados.

Em caso de sucesso, a saída será dominada por mensagens com a palavra-chave PASS, demonstrando que todas as suítes foram bem-sucedidas. Se algum teste falhar, o Jest destaca o erro em vermelho, informando qual componente falhou, qual teste específico apresentou problema e uma descrição clara da falha.

Por exemplo, se um componente espera que determinado elemento esteja presente no DOM, mas este não é renderizado corretamente, o Jest exibirá uma mensagem de erro indicando que não foi possível encontrar o elemento esperado, junto à linha de código de teste que falhou. Com o uso da extensão `@testing-library/jest-dom`, diversos matchers adicionais são disponibilizados, como `toBeInTheDocument()`, que permite verificar com precisão se um elemento está presente na árvore do DOM renderizada, facilitando a identificação de erros e tornando os testes mais legíveis.

Esse mecanismo de execução e feedback rápido é fundamental para processos de desenvolvimento ágeis, permitindo que falhas sejam detectadas e corrigidas de forma imediata, preservando a qualidade e prevenindo regressões ao longo do tempo.

Conclusão

A abordagem adotada para a implementação dos testes unitários neste projeto seguiu as melhores práticas reconhecidas na indústria de desenvolvimento de software, utilizando ferramentas maduras, confiáveis e adequadas ao contexto de uma aplicação moderna baseada em React e TypeScript. A configuração meticulosa do ambiente de testes, aliada à filosofia de testes orientados ao comportamento do usuário, resultou em uma suíte de testes eficiente e sustentável.

Esse conjunto de práticas fortalece a robustez do sistema, promove segurança na evolução do código e proporciona uma base sólida para o desenvolvimento contínuo da aplicação. Além disso, estabelece um padrão para a introdução futura de testes de integração e testes end-to-end, elevando ainda mais o nível de qualidade e confiabilidade do projeto.

Referências

JEST. *Jest Documentation*. Disponível em: <https://jestjs.io>. Acesso em: 20 maio 2025.

DOM-TESTING-LIBRARY. *React Testing Library Documentation*. Disponível em: <https://testing-library.com/docs/react-testing-library/intro>. Acesso em: 20 maio 2025.

TS-JEST. *ts-jest Documentation*. Disponível em: <https://kulshekhar.github.io/ts-jest>. Acesso em: 20 maio 2025.

IDENTITY-OBJ-PROXY. *identity-obj-proxy Documentation*. Disponível em: <https://github.com/keyz/identity-obj-proxy>. Acesso em: 20 maio 2025.

MESZAROS, Gerard. *xUnit test patterns: refactoring test code*. Boston: Addison-Wesley, 2007.