Universidade Federal de Pernambuco

Centro de informática

Curso de Engenharia da Computação

Gerenciamento de Estados no React JS: Uma Avaliação de Desempenho entre Redux e Context API

Trabalho de Conclusão de Curso de Graduação

por

Washington Igor dos Santos Silva

Orientador: Prof. Eduardo Antonio Guimaraes Tavares

Recife, Outubro / 2024

Washington Igor dos Santos Silva

Gerenciamento de Estados no React JS: Uma Avaliação de Desempenho entre Redux e Context API

Monografia apresentada ao Curso de Engenharia da Computação, como requisito parcial para a obtenção do Título de Bacharel em Engenharia da Computação, Centro de informática da Universidade Federal de Pernambuco.

Orientador: Prof. Eduardo Antonio Guimaraes Tavares

Recife

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Washington Igor dos Santos.
Gerenciamento de Estados no React JS: Uma Avaliação de Desempenho entre Redux e Context API / Washington Igor dos Santos Silva. - Recife, 2024.
48 : il., tab.

Orientador(a): Eduardo Antonio Guimaraes Tavares
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2024.
Inclui referências, apêndices.

1. React JS. 2. Web Front-end. 3. Redux. 4. Context API. I. Guimaraes Tavares, Eduardo Antonio . (Orientação). II. Título.

000 CDD (22.ed.)

2024

WASHINGTON IGOR DOS SANTOS SILVA

GERENCIAMENTO DE ESTADOS NO REACT JS: Uma Avaliação de Desempenho entre Redux e Context API

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação. .

Aprovado em: 13/08/2024

BANCA EXAMINADORA

Prof. Dr. Eduardo Antonio Guimaraes Tavares (Orientador)
Universidade Federal de Pernambuco

Prof. Dr. Jamilson Ramalho Dantas (Examinador Interno)
Universidade Federal de Pernambuco

Agradecimentos

Eu gostaria de agradeçer a todos vocês que me ajudaram durante esta jornada, especialmente para:

Minha companheira Carine, a família que a vida me deu, Kássia e Nathália.

Meus amigos da faculdade, os quais tenho grupos incríveis no WhatsApp, sendo eles 'Comunistas Desistentes EC' e 'Otakus e os pai do Celta'

Um agradecimento especial ao meu professor orientador, que desde 2019 faz parte da minha vida acadêmica e ao meu grande amigo Ian Karlos.

Trabalho duro é inútil para aqueles
que não acreditam em si mesmos.

Naruto Uzumaki

RESUMO

Este trabalho apresenta uma avaliação comparativa entre Redux e Context API para gerenciamento de estado no React JS, focando no tempo de execução e no consumo de memória. Para a coleta de dados, foi utilizada a biblioteca Puppeteer, uma ferramenta Node.js que permite executar um navegador Chrome/Chromium, possibilitando a abertura de páginas web, a simulação de interações do usuário e a coleta de métricas de desempenho. A metodologia utilizada inclui análise estatística com o *Teste T de Student*. Os resultados oferecem percepções sobre a eficiência de cada ferramenta, auxiliando na escolha adequada para projetos de desenvolvimento web.

Palavras-chave: Redux, Context API, React JS, gerenciamento de estado, desempenho, tempo de execução, consumo de memória

ABSTRACT

This work presents a comparative evaluation between Redux and Context API for state management in React JS, focusing on execution time and memory consumption. For data collection, the Puppeteer library was used, a Node.js tool that allows running a Chrome/Chromium browser, enabling the opening of web pages, the simulation of user interactions and the collection of performance metrics. The methodology used includes statistical analysis with the Student's T Test. The results offer insights into the efficiency of each tool, helping to choose the right one for web development projects.

Keywords: Redux, Context API, React JS, state management, performance, execution time, memory consumption

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

UFPE	Universidade Federal de Pernambuco
UI	User Interface
MVC	Model-View-Controller
JSX	JavaScript XML
API	Application Programming Interface
HTML	HyperText Markup Language
DOM	Document Object Model
SEO	Search Engine Optimization

SUMÁRIO

1 INTRODUÇÃO

No contexto dinâmico e sempre evoluindo da web, as tendências e tecnologias desempenham um papel crucial na definição do panorama do desenvolvimento de software. Em maio de 2023, ocorreu o Developer Survey 2023, organizado pelo *Stack Overflow*, um dos maiores fóruns online de tecnologia do mundo. O levantamento contou com mais de 67 mil respostas de desenvolvedores profissionais. O JavaScript foi apontado como a linguagem de programação mais popular. Na categoria de frameworks e tecnologias web, a biblioteca React, desenvolvida em JavaScript, também conquistou o primeiro lugar na avaliação dos profissionais [1].

O React é uma biblioteca JavaScript de código aberto, que é um conjunto de linhas de código JavaScript pré-prontos. O React implementa princícios de programação funcional e design de componentes, que são pontos importantes na engenharia de software moderna para a criação de interfaces de usuário(*user interfaces* - UI) eficientes e escaláveis. Por adotar uma abordagem declarativa, o React permite que os desenvolvedores especifiquem o que a interface deve representar, em vez de como as atualizações devem ocorrer, abstraindo a complexidade de desenvolvimento de aplicações.om o React é possível dividir uma página inteira em partes, assim sendo possível trabalhar em cada parte de forma individual e independente. Outro fator interessante é que as partes podem ser reaproveitadas entre si, o que facilita e otmiza o desenvolvimento de uma UI [2].

1.1 História

O React teve seu primeiro protótipo criado em 2011 por Jordan Walke, Engenheiro de Software do Facebook. Em 2013, durante a JS ConfUS, o React foi apresentado e lançado como código aberto. Logo após sua divulgação, o React começou a ser experimentado na indústria e, em 2015, foi considerado 'Estável'. O notável crescimento do React pode ser atribuído à extensa gama de ferramentas e recursos que facilitam a construção e manutenção de UIs, sendo um dos principais o gerenciamento avançado de estado. Juntamente com seu modelo de documento do objeto(*Document Object Model* - DOM) virtual, utilizado para otimizar as atualizações do DOM real, isso favorece o desempenho de aplicações que requerem atualizações frequentes em sua interface [3].

Em 2015, o conceito de estado no React foi revolucionado quando Dan Abramov

lançou o Redux, uma biblioteca JavaScript inspirada no padrão de gerenciamento de estado Flux, que consiste em uma arquitetura de fluxo unidirecional de dados, criado pelo Facebook em 2014 para enfrentar os desafios de gerenciar o estado da aplicação em grandes projetos de front-end. O Redux rapidamente se tornou a ferramenta de gerenciamento de estado mais utilizada no React, pois até então, o React possuía apenas sua interface de programação de aplicação (*Application Programming Interface* - API) de contexto interna, que neste momento possuia erros conhecidos na parte de trasmissão de valores atualizados, o que facilitou a adoção do Redux como forma de gerenciamento de estado [4]. No entanto, em março de 2018, na versão 16.3.0 do React, foi introduzida sua nova API de contexto, proporcionando aos desenvolvedores uma ferramenta nativa mais robusta para gerenciar suas aplicações de forma eficiente [5].

1.2 **Motivação**

O maior desafio em um aplicativo React é o gerenciamento de estado, onde o estado representa os dados do contexto de execução da aplicação, como por exemplo, as informações do usuário logado. Em aplicações grandes, o React por si só não é suficiente para lidar com a complexidade de compartilhar dados entre diferentes níveis da árvore de componentes, o que leva os desenvolvedores a usarem métodos de gerenciamento de estado como a Context API e Redux. Entender o desempenho de cada método, se torna um fator importante na hora de decidir qual dos meios utilizar em uma aplicação. [6].

1.3 **Objetivos**

Neste projeto, será conduzida uma comparação entre as duas ferramentas, Context API e Redux, explorando dois aspectos essenciais: o tempo de execução e o consumo de memória. O intuito é fornecer ao leitor informações relevantes para ajudá-lo a decidir qual abordagem adotar ao precisar gerenciar com qualidade os estados de sua aplicação React. A comparação irá se dar por testes de hipótese formulados usando o tempo de execução e o consumo de memória para as diferentes ferramentas.

1.4 **Organização**

Nos próximos capítulos, serão explicados de forma mais detalhada os elementos centrais discutidos aqui. O capítulo 2 traz mais detalhes sobre o que é o React e o seu gerenciamento de estado, além de como funcionam as ferramentas Context API e Redux. O capítulo 3 aborda a metodologia aplicada para tornar mais compreensível o caminho percorrido para obter os resultados experimentais contidos no capítulo 4. Por fim, o capítulo 5 traz uma breve discussão acerca dos resultados obtidos e percepções sobre o uso adequado do Context API e do Redux.

2 REACT E GERENCIAMENTO DE ESTADO

Neste capítulo, é explicado com mais detalhes como o React funciona, o que é seu gerenciamento de estado e como as ferramentas Context API e Redux podem auxiliar aplicações a ter um bom gerenciamento e organizar o fluxo de dados interno.

2.1 React

O React é uma biblioteca JavaScript projetada para renderizar UI. Essas interfaces são construídas a partir de unidades menores, como botões, texto e imagens. O React possibilita a combinação dessas unidades para formar um componente, um bloco de código que pode ser reutilizado em toda a aplicação [7]. No contexto do React, as telas são compostas por aninhamentos de componentes, resultando em uma árvore de componentes conhecida como DOM virtual. Muitos desenvolvedores utilizam o React como a camada de visualização (V) no padrão de arquitetura de software Modelo-Visão-Controle (*Model-View-Controller* - MVC) que é focado no reuso de código. Devido à abstração do DOM pelo React, é possível alcançar um modelo de programação mais simples e um melhor desempenho [8].

Um aspecto fundamental do React é o seu fluxo de dados unidirecional, implementado através do padrão Flux. Esse padrão assegura que os dados tenham apenas uma maneira de serem transferidos de um componente para outro, simplificando a gestão do estado da aplicação. Na Figura 1 temos os componentes utilizados no Flux. De maneira simples, o *Action* manipula os eventos gerados pelas interações do usuário, após ser realizada, a *Action* é despachada usando o *Dispatcher*, ele é o responsável por levar o conteúdo proveniente de uma *Action* para a Store e assim a atualizar, como também controla o tráfego de atualizações, garantindo que nenhuma nova *Action* aconteça antes da *Store* ser atualizada. A *Store* contém o estado da aplicação, uma aplicação pode ter mais de uma *Store*, pois elas são uma camada do padrão Flux, a *Store* é responsável por avisar a *View* que houve uma alteração e assim a *View* que é o responsável por renderizar a UI, tem uma nova renderização e assim atualizando o seu conteúdo [9].

Alguns dos recursos do React incluem o JSX, Componentes e *Hooks* [10].

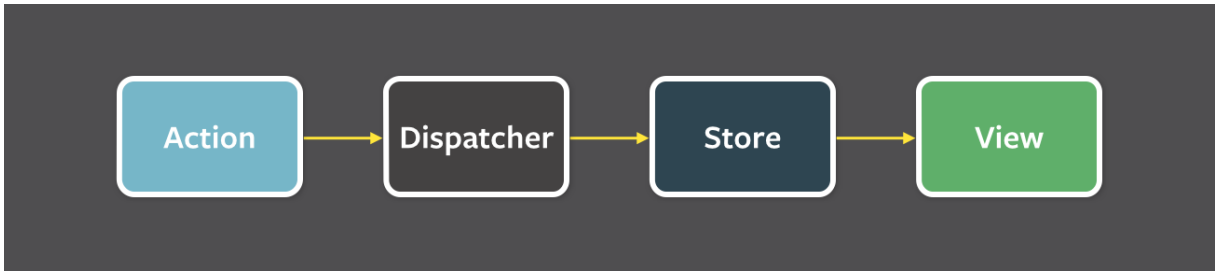


Figura 1: O padrão Flux.
Fonte: https://www.freecodecamp.org/news/how-to-use-flux-in-react-example/.

2.1.1 JSX

O JSX, que significa JavaScript XML, é uma extensão de sintaxe para JavaScript, que permite escrever marcações semelhantes a HTML que é uma linguagem de marcação de hipertexto(*HyperText Markup Language* - HTML), dentro de um arquivo JavaScript [11]. Na Figura 2, temos um exemplo simples de uso do JSX.

O JSX possui regras próprias [12]:

- Retornar um único elemento raiz: O JSX se assemelha ao HTML, mas, por baixo, é transformado em objetos JavaScript simples. Não é possível retornar dois objetos de uma função sem agrupá-los em um array, o que justifica a necessidade de agrupar duas tags JSX em outra tag ou fragmento [12].

- Fechar todas as tags: O JSX exige que as tags sejam explicitamente fechadas [12].

- Modelo *camelCase* em todas as coisas: O modelo *camelCase* é uma prática de escrita onde se usa palavras compostas, onde cada palavra é iniciada com maiúscula, assim como o próprio modelo, *camelCase* [13]. O JSX se transforma em JavaScript, e este possui limitações nos nomes de variáveis. Por exemplo, os nomes não podem conter hifens ou palavras reservadas. No React, muitos atributos do HTML são escritos em *camelCase*. Alguns elementos fogem dessa regra por razões internas do próprio React, como o atributo aria-*, que é escrito com hífen [12].

2.1.2 Componentes

Os componentes representam um dos conceitos fundamentais do React, constituindo a base das UI. Na abordagem tradicional da criação de páginas web, os desen-

```jsx
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

Figura 2: Um exemplo de uso do JSX.
Fonte: https://pt-br.legacy.reactjs.org/docs/introducing-jsx.html.

volvedores estruturam o conteúdo com HTML e adicionam interações de usuário usando JavaScript. No entanto, essa abordagem funcionava bem quando as interações não eram consideradas essenciais. Hoje em dia, espera-se que as aplicações sejam altamente interativas. O React concentra-se fortemente na interatividade, e um componente React, por definição, é uma função JavaScript que permite a adição de tags HTML. Dessa forma, o controle da interação pode ser realizado de maneira isolada e mais precisa[9]. No React temos componentes de função e classe, neste documento iremos nos manter utilizando apenas componentes de função [12].

2.1.3 Hooks

Os *Hooks* foram adicionados ao React na versão 16.8, trazendo com eles a possibilidade de usar o *state* e outros recursos, sem escrever componentes de classe. A introdução dos *Hooks* não acabou com o uso de componentes de classe no React, apenas permitiu que os componentes de função pudessem utilizar os recursos do React [12].

2.2 Gerenciamento de estado no React

Em aplicações React, a forma mais comum de transmitir dados entre componentes é utilizando propriedades. No entanto, quando esses componentes estão em diferentes níveis dentro da árvore de componentes, torna-se necessário passar essas propriedades por meio de componentes intermediários. Isso resulta na escrita de código adicional e na entrega de propriedades a componentes que podem não utilizar esses dados para nada além de repassá-los para o próximo nível da árvore de componentes. Essa prática impacta diretamente no desempenho e na manutenção da aplicação. Na Figura 3, é possível observar essa comunicação, onde um componente filho mais abaixo na árvore precisa acessar uma informação provida pelo componente pai mais acima na árvore, resultando na necessidade de passar o dado por diferentes componentes intermediários [14].

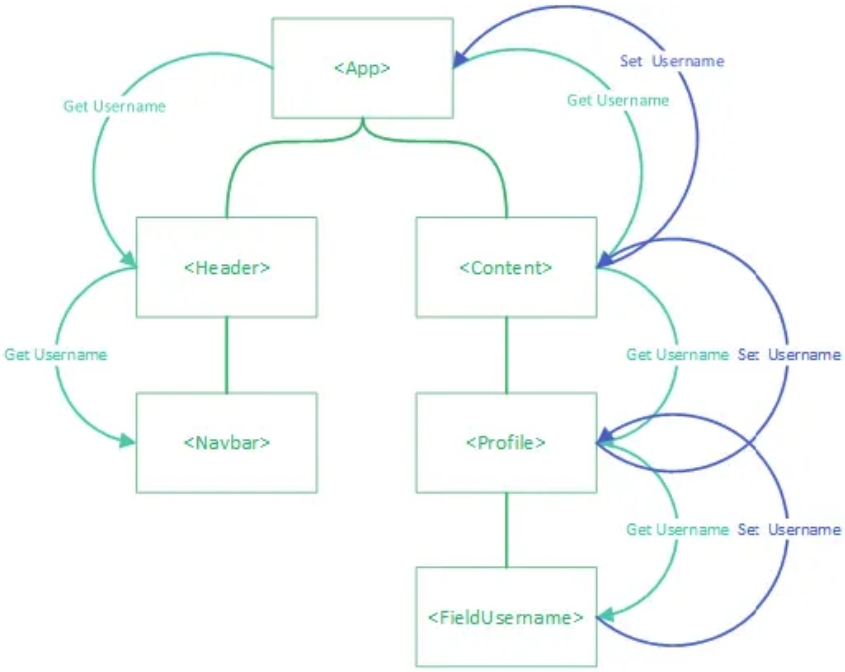


Figura 3: Passagem de propriedades entre componentes em uma árvore de componentes React.
Fonte: https://medium.com/loftbr/escalando-o-gerenciamento-de-estados-no-react-7704dab2d56d.

Para lidar com esse problema, desenvolvedores criaram diversas bibliotecas JavaScript, como o Redux e, internamente, o próprio React em sua versão 16.3.0, que introduziu a Context API. Ambas são capazes de oferecer soluções para o gerenciamento de estado global [4, 5].

2.3 Redux

Inspirado pelo padrão Flux, Dan Abramov criou a biblioteca Redux em 2015, com o objetivo inicial de apresentar durante uma palestra a "depuração de viagem no tempo", que consiste em uma forma de capturar um rastro do seu processo à medida que ele é executado e, em seguida, reproduzi-lo mais tarde, tanto para frente quanto para trás. Projetada para seguir os princípios do padrão Flux, a biblioteca incorporou conceitos de programação funcional em sua implementação. Essa abordagem possibilitava a análise do estado em diferentes momentos do ciclo de vida da aplicação, ao mesmo tempo em que tornava o código mais direto, testável e compreensível. Lançado no mesmo ano, o Redux rapidamente ganhou destaque, superando outras bibliotecas inspiradas no Flux [4].

O Redux pode ser integrado ao React ou a qualquer outra biblioteca de visualiza-

ção. Além disso, oferece um amplo ecossistema de complementos disponíveis [15].

Devido ao fato de o Redux seguir o fluxo de dados unidirecional, é crucial ter um controle sobre o estado da aplicação. Conforme a aplicação cresce e se torna mais complexa, a ausência desse controle pode tornar mais difícil a reprodução de problemas e a adição de novos recursos. O Redux simplifica a complexidade do código, mantendo as atualizações de estado sob controle, o que facilita o gerenciamento de estados atualizados. A Figura 4 ilustra o fluxo de dados Redux, destacando os três pontos principais: *Actions*, *Reducer* e *Store*. O fluxo pode ser descrito da seguinte forma [16]:

- Uma *Action* é despachada quando um usuário interage com a aplicação [16].
- Uma função do *Reducer* raiz é chamada com o estado atual e a *Action* despachada. O *Reducer* pode dividir a tarefa entre funções menores e, por fim, irá retornar um novo estado [16].
- A *Store* notifica a *View* sobre a mudança de estado [16].
- A *View* pode recuperar o estado atualizado e renderizar novamente [16].

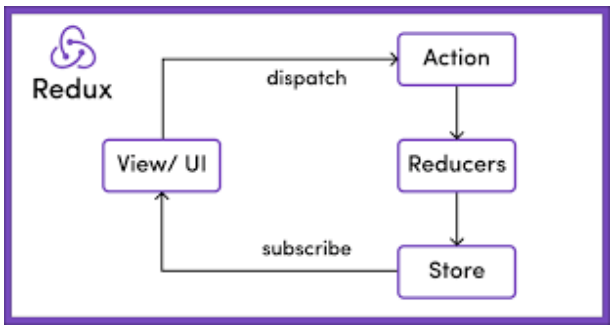


Figura 4: Fluxo de dados Redux.
Fonte: `https://dev.to/nickgabe/redux-toolkit-e-react-redux-2o9k`.

2.3.1 Actions

As *Actions* são as fontes de informação enviadas da aplicação para a *Store* por meio dos *Reducers*. Elas são disparadas por meio de *Action creators*, que são funções puras responsáveis por criar as *Actions*. Uma *Action* é um objeto que obrigatoriamente possui um atributo chamado *"type"*, indicando qual ação está sendo despachada, podendo

também conter dados associados. Caso existam dados associados a uma *Action*, os mesmos são carregados por meio de um atributo chamado *"payload"* [17].

2.3.2 Reducers

Os *Reducers* são funções puras, ou seja, funções que não geram efeitos colaterais, garantindo que uma mesma entrada resulte em uma mesma saída. Sua funcionalidade é disparar eventos e evoluir atributos da *Store*, manipulando o estado global da aplicação [17].

Podemos entender um *Reducer* como um filtro, recebendo e processando informações, em seguida enviando esses dados à *Store*. Eles são responsáveis por lidar com as *Actions* provenientes das interações na aplicação. Por padrão, é recomendado que para cada dado contido na *Store*, exista um *Reducer* correspondente [17].

Todo *Reducer* escuta todas as *Actions* disparadas pela aplicação, e a *Store* só é modificada por meio das manipulações realizadas pelos *Reducers* [17].

2.3.3 Store

A *Store* é um container imutável, o que significa que não sofre alterações, apenas evoluções. Seu papel fundamental é armazenar e centralizar o estado global da aplicação, representando o conjunto de estados da mesma [17].

Um dos princípios fundamentais que define o Redux é "Um único ponto de verdade", e esse princípio é rigorosamente seguido pela *Store*. Além disso, a *Store* adere a outros princípios, como a imutabilidade do estado, garantindo que o estado da aplicação seja inalterável, mas evoluível. As alterações são realizadas exclusivamente por funções puras [17].

Por fim, a *Store* assume a responsabilidade de monitorar e notificar as mudanças que ocorrem no estado, informando as entidades que precisam estar cientes dessas alterações [17].

2.4 Context API

Passar propriedades que são utilizadas por muitos componentes dentro da aplicação, como por exemplo, o tema da UI, se torna complicado conforme a aplicação cresce.

Com a Context API, o React disponibilizou uma forma de compartilhar esses dados entre todos os componentes da mesma árvore de componentes, sem precisar passar explicitamente as propriedades entre pai e filho a cada nível da árvore [18].

O uso da Context API consiste em dois componentes principais: O provedor de contexto e o consumidor de contexto. O provedor é o responsável por criar e gerenciar o contexto, o qual possui os dados a serem compartilhados. O consumidor é utilizado para acessar o contexto e seus dados dentro de um componente [19].

Para criação de um contexto na aplicação, alguns passos simples precisaram ser seguidos [19]:

1. Criar o objeto de contexto: O React possui uma função chamada de *'createContext'*, com ela é criado um objeto de contexto, o qual conterá os dados que é desejado compartilhar na aplicação [19]. A Figura 5, mostra um exemplo do uso da função.

```
import { createContext } from 'react';

export const MyContext = createContext("");
```

Figura 5: Criando um novo contexto.
Fonte: https://www.freecodecamp.org/news/context-api-in-react/.

2. Envolver com o provedor os componentes que precisam ter acesso aos dados: O componente *Provider* aceita uma propriedade *"value"* que contém os dados compartilhados, qualquer componente filho do componente *Provider* na árvore de componentes, poderá acessar os dados compartilhados. A Figura 6, mostra o uso de um componente *Provider*, nesse exemplo temos o *Provider* recebendo os valores de estado *'text'* e uma função de evolução de estado *'setText'*, o componente filho *'MyComponent'* terá internamente acesso a esses valores [19].

3. Consumir o contexto: Após a criação do contexto e de seu provedor, pode-se consumir seus dados, para isso, o React possui um *hook* nomeado de *'useContext'*. Na Figura 7, o *useContext* é usado para acessar os valores *'text'* e *'setText'* que foram atribuídos ao contexto como visto na Figura 5. Dentro da instrução de *return* do componente *"MyComponent"*, é renderizado um elemento de parágrafo que exibe o valor de *'text'*. Também é renderizado um botão para alterar o valor de *'text'*

para *'Hello, world'*, pois quando clicado está programado para acionar a função *'setText'* [19].

```
// Create a parent component that wraps child components with a Provider

import { useState, React } from "react";
import { MyContext } from "./MyContext";
import MyComponent from "./MyComponent";

function App() {
  const [text, setText] = useState("");

  return (
    <div>
      <MyContext.Provider value={{ text, setText }}>
        <MyComponent />
      </MyContext.Provider>
    </div>
  );
}

export default App;
```

Figura 6: Criando o provider.
Fonte: https://www.freecodecamp.org/news/context-api-in-react/.

```
import { useContext } from 'react';
import { MyContext } from './MyContext';

function MyComponent() {
  const { text, setText } = useContext(MyContext);

  return (
    <div>
      <h1>{text}</h1>
      <button onClick={() => setText('Hello, world!')}>
        Click me
      </button>
    </div>
  );
}

export default MyComponent;
```

Figura 7: Criando um consumidor do contexto.
Fonte: https://www.freecodecamp.org/news/context-api-in-react/.

Seguindo os passos anteriores, tem-se um contexto, que ao envolver uma parte dos componentes da árvore de componentes, vai permitir que os dados disponíveis no objeto de contexto sejam acessados por qualquer componente desta árvore de forma simples e eficiente [19].

3 METODOLOGIA

Neste capítulo, é explicado a metodologia aplicada ao trabalho, que tem como objetivo avaliar o desempenho das ferramentas Redux e Context API no gerenciamento de estados em aplicações React. Inicialmente, foram definidos os sistemas que seriam utilizados para os experimentos. Duas versões de cada sistema foram desenvolvidas: uma com Redux e outra com Context API. Após isso, foram definidas as métricas que serão avaliadas e aplicadas nas hipóteses para análise estatística.

No projeto de experimentos, foram definidos os passos que seriam seguidos para realizar a coleta de dados de forma a satisfazer os conceitos estatísticos empregados, como o teorema do limite central.

As etapas a seguir detalham cada passo da metodologia, seguindo o fluxo visto na Figura 8.

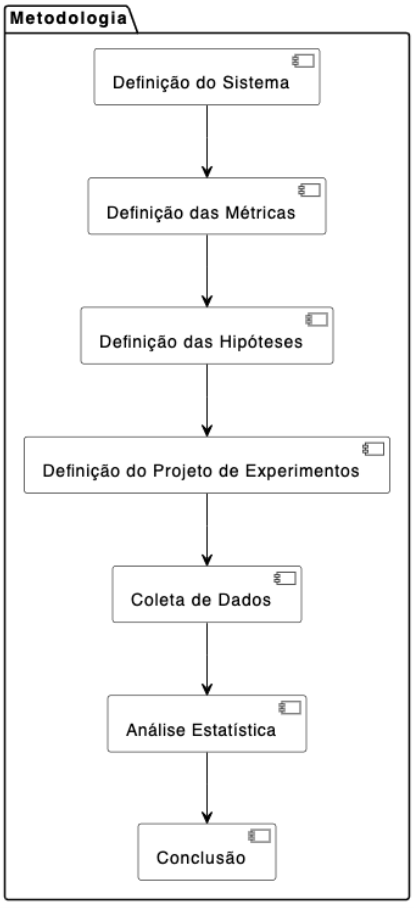


Figura 8: Fluxograma da metodologia aplicada.
Fonte: https://online.visual-paradigm.com/app/diagrams/.

3.1 Definição do sistema

Para a experimentação, foram criados dois sistemas. O primeiro, visível na Figura 9 consiste em uma aplicação web que permite a criação de *cards*[1]. Cada *card* é composto por um título, um subtítulo e dois botões[2]. Um desses botões é destinado à exclusão, apagando o *card*, enquanto o outro abre um *modal*[3] para a edição dos campos de título e subtítulo. O *modal* apresenta dois *inputs*[4], que são campos de entrada do usuário, representando os campos correspondentes, além de dois botões. O primeiro botão salva as edições realizadas, enquanto o segundo cancela a edição. Essa estrutura proporciona uma maneira intuitiva e eficiente de criar e modificar os *cards* na aplicação.

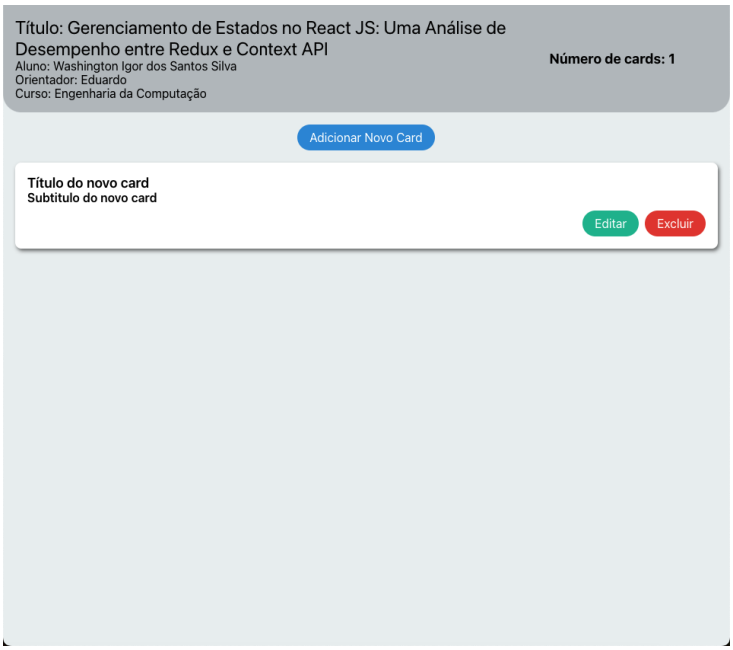


Figura 9: Aplicação de criação de cards.
Fonte: .

O segundo sistema, visível na Figura 10 também uma aplicação web, é constituído por um *select*[5] e uma composição de quatro *cards* que ficam sobrepostos, cada um com uma cor própria. O *select* permite a mudança de tema da aplicação, alterando assim a cor dos *cards*.

[1] O que é um card? Disponível em https://goalfy.com.br/academy/o-que-e-um-card/
[2] O que é um botão? Disponível em https://productoversee.com/botoes-em-ui-design/
[3] O que é um modal? Disponível em https://blog.hubspot.com/website/modal-web-design
[4] O que é um input? Disponível https://www.significados.com.br/input/
[5] O que é um select? Disponível em https://blog.hubspot.com/website/html-select

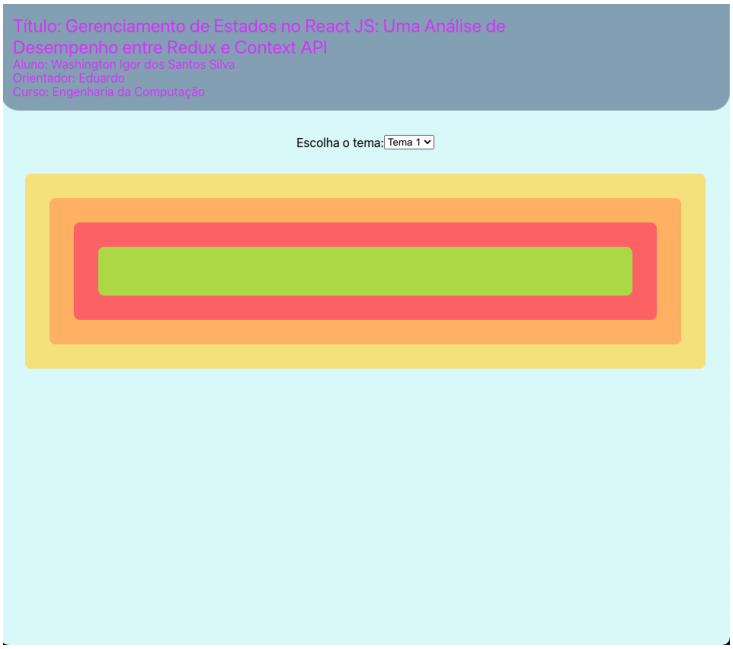


Figura 10: Aplicação de mudança de tema.
Fonte: .

3.2 Definição das métricas

Um tempo de execução baixo é um fator importante na experiência do usuário, reduzir o tempo de carregamento e resposta das páginas, pode aumentar a conversão devido a melhor experiência. Outro fator importante sobre o tempo de execução, é que os motores de busca (*Search Engine Optimization* - SEO), consideram o tempo de carregamento como um fator importante na classificação de um site, afetando a sua visibilidade diretamente.

O consumo de memória excessivo, pode levar os sites a terem instabilidades, travamentos e até fechamentos forçados controlados pelo próprio navegador, isso impacta na confiabilidade do mesmo. Uma gestão eficaz de memória, permite que uma melhor escalabilidade e desempenho nas aplicações.

Por se tratar de dois fatores importantes e diretamente ligados com o gerenciamento de estado, o tempo de execução e o consumo de memória, foram as métricas definidas para a análise de desempenho, pois uma aplicação com bom gerenciamento, tende a consumir menos memória e executar suas atividades mais rapidamente.

3.3 Hipóteses

Com base nas métricas analisadas, foram formuladas duas hipóteses, sendo elas:

1. O tempo de execução da Context API é significativamente menor do que o tempo de execução do Redux.

   - **Hipótese Nula ($H_{0,\textbf{tempo}}$):**

   $$H_{0,\text{tempo}} : \mu_{\text{Context API, tempo}} = \mu_{\text{Redux, tempo}}$$

   - **Hipótese Alternativa ($H_{a,\textbf{tempo}}$): Teste unilateral à esquerda**

   $$H_{a,\text{tempo}} : \mu_{\text{Context API, tempo}} < \mu_{\text{Redux, tempo}}$$

   Onde:

   - $\mu_{\text{Context API, tempo}}$: Média do tempo de execução da Context API
   - $\mu_{\text{Redux, tempo}}$: Média do tempo de execução do Redux

2. O consumo de memória da Context API é significativamente menor do que o consumo de memória do Redux.

   - **Hipótese Nula ($H_{0,\textbf{memória}}$):**

   $$H_{0,\text{memória}} : \mu_{\text{Context API, memória}} = \mu_{\text{Redux, memória}}$$

   - **Hipótese Alternativa ($H_{a,\textbf{memória}}$): Teste unilateral à esquerda**

   $$H_{a,\text{memória}} : \mu_{\text{Context API, memória}} < \mu_{\text{Redux, memória}}$$

   Onde:

   - $\mu_{\text{Context API, memória}}$: Média do consumo de memória da Context API
   - $\mu_{\text{Redux, memória}}$: Média do consumo de memória do Redux

3.4 Projeto de Experimentos

Para a verificação das hipóteses propostas, foram planejados dois experimentos em ambiente controlado. Cada experimento utilizou um dos sistemas definidos no projeto, o fator considerado foi o Gerenciamento de Estado em aplicações React e o níveis correspondentes desse fator, foram as ferramentas Redux e Context API. A seguir, está a estrutura mais detalhada para cada experimento.

3.4.1 Experimento - Criação de Cards

Para o experimento de Criação de Cards, o fator Gerenciamento de Estado em aplicações React é aplicado de forma mais evidente, quando temos a interação do usuário com os botões da tela e os componentes de entrada. As ferramentas Redux e Context API que representam os fatores, vão impactar diretamente no tempo de resposta dessas ações e no custo de memória consumida para as realizar. Para esse experimento, temos os seguintes cenários para analisar:

- No primeiro cenário será analisado o tempo de execução da aplicação e o seu consumo de memória para o cenário onde o gerenciamento de estado será feito utilizando o Redux.
  - **Fator:** Gerenciamento de Estado
  - **Nível:** Redux
  - **Medidas de Desempenho:**
    - Tempo de Execução
    - Consumo de Memória
- No segundo cenário será analisado o tempo de execução da aplicação e o seu consumo de memória para o cenário onde o gerenciamento de estado será feito utilizando o Context API.
  - **Fator:** Gerenciamento de Estado
  - **Nível:** Context API
  - **Medidas de Desempenho:**

  - Tempo de Execução
  - Consumo de Memória

3.4.2 Experimento - Mudança de tema

Para o experimento de mudanças de tema, o fator Gerenciamento de Estado em aplicações React é aplicado de forma mais evidente, quando o usuário selecionar a troca de tema, onde todas as cores em tela são alteradas, com isso, cada elemento precisa saber qual é a sua nova cor, os *cards* centrais, são componentes de um mesmo ramo da árvore de componentes, de forma que o *card* mais interno, é filho do primeiro *card* acima dele, que por sua vez, também é filho do *card* acima dele e assim por diante, até o *card* mais afastado que é o primeiro da sequência. Em um ambiente sem uma gerenciamento de estados adequado, é provável que as cores seriam passadas entre os filhos, a cada nível da árvore de componentes, o que levaria um dos problemas levantados aqui anteriormente, a escrita desnecessária de código. Com as ferramentas Redux e Context API, esse gerenciamento de estados é feito de forma mais eficaz, além de tornar o código mais legível, ele também vai impactar diretamente o consumo de memória, pois vai ser necessário menos código sendo passado entre os componentes para alcançar os níveis mais baixos da árvore de componentes e o tempo que os componentes vão precisar para atualizar em resposta a ação do usuário em trocar o tema. Para esse experimento, temos os seguintes cenários para analisar:

- No primeiro cenário será analisado o tempo de execução da aplicação e o seu consumo de memória para o cenário onde o gerenciamento de estado será feito utilizando o Redux.
  - **Fator:** Gerenciamento de Estado
  - **Nível:** Redux
  - **Medidas de Desempenho:**
    - Tempo de Execução
    - Consumo de Memória
- No segundo cenário será analisado o tempo de execução da aplicação e o seu consumo de memória para o cenário onde o gerenciamento de estado será feito

utilizando o Context API.

- **Fator:** Gerenciamento de Estado
- **Nível:** Context API
- **Medidas de Desempenho:**
  - Tempo de Execução
  - Consumo de Memória

3.5 Coleta de Dados

Para a coleta dos dados dos experimentos, foi utilizado o Puppeteer. O Puppetter é uma biblioteca feita em JavaScript que fornece uma forma fácil para controlar ações em um navegador Chrome/Chromium através do protocolo DevTools, disponibilizado pelo Google. Por meio do Puppeteer é possível criar um ambiente de teste automatizado usando os recursos mais recentes de JavaScript e navegador, dessa forma podemos simular as ações de um usuário em determinada página, seguindo um script pré definido, fazendo com que as repetições dos testes, estejam livres de interferências humanas. Com ele também é possível capturar um rastreamento da linha do tempo da aplicação para ajudar a diagnosticar problemas de desempenho [20]. A linha do tempo capturada, pode ser utilizada dentro do Google DevTools, uma ferramenta do Google que utiliza o protocolo DevTools [21], dentro da ferramenta, a linha do tempo pode ser vista no seu painel de desempenho [22], o qual, exibe as medidas de desempenho requisitadas nos experimentos. O Puppeteer também retorna tais informações dentro um método próprio, o 'metrics' [23], o quão será utilizado para coleta das informações.

Utilizando o Google DevTools e o Puppeteer, foram coletadas 50 amostras de cada experimento, devido ao teorema do limite central [24], o qual é um fator importante para os conceitos que serão empregados.

3.6 Análise Estatística

Para análise estatística, foi escolhido utilizar o *Teste T de Student* para comparar a média de duas amostras independentes, para tal, foi realizada a verificação da normalidade dos dados, já que o Teste T é robusto em relação à normalidade dos dados,

especialmente com tamanhos de amostra grandes. Com $n = 50$, que é a quantidade de amostras coletadas, a condição de normalidade geralmente é razoavelmente aceitável.

Com o método devidamente apresentado, junto aos dados coletados, o Teste T padrão foi utilizado para a análise estatística.

Assim segue o passo a passo executado para experimentação:

1. Organização dos dados coletados
2. Teste de normalidade dos dados, usando *Ryan-Joiner test*
3. Definição do nível de significância (α)
    - As análises realizadas, consideraram o valor $\alpha = 0.01$, devido ao teste de normalidade
4. Cálculo do Teste Estatístico
    - Encontrar o valor do *Test T* apropriado
5. Cálculo do Valor-p
    - Determinar o Valor-p associado ao *Test T* calculado.
6. Comparação do Valor-p com o nível de significância α
7. Interpretação do resultado obtido

3.7 Conclusão

Por meios dos experimentos, foram coletados dados em quantidades suficientes, que ao passar no teste de normalidade de *Ryan-Joiner test*, possibilitou a aplicação do *Test T de Student*. Utilizando esse método, os valores resultantes foram devidamente associados aos Valores-p correspondentes. Com esses dados, realizou-se a comparação com o nível de significância α, fornecendo resultados adequados para a interpretação.

4 RESULTADOS EXPERIMENTAIS

Esta seção apresenta os resultados dos experimentos realizados para comparar o desempenho entre Redux e Context API no gerenciamento de estados em aplicações web utilizando React. As métricas analisadas para discussão dos resultados foram o tempo de execução, que corresponde ao período em que o sistema estava ativo para realizar tarefas na aplicação, e o uso de memória do JavaScript para alocação de dados necessários para o funcionamento da aplicação durante o ciclo de vida.

4.1 Configuração do Ambiente de avaliação

Os testes foram realizados em um ambiente de desenvolvimento configurado conforme descrito nas Tabela 1 e Tabela 2. Para reduzir ruídos e interferências, o ambiente foi limpo de quaisquer processos não essenciais.

Tabela 1: Configuração de Hardware do Ambiente de Avaliação

Hardware	
Componente	**Especificação**
CPU	AMD Ryzen 5 5600X 6-Core Processor 3.70 GHz
RAM	32GB DDR4
Armazenamento	SSD 500GB

Tabela 2: Configuração de Software do Ambiente de Avaliação

Software	
Componente	**Especificação**
Sistema Operacional	Windows 10 Pro
Node.js	v20.12.2
React	v18.2.0
Redux(@reduxjs/toolkit)	v2.2.1
Puppeteer	v22.9.0
Navegador	HeadlessChrome/125.0.6422.60

4.2 Experimentos Realizados

Para a realização de uma avaliação em cima dos experimentos, foram construidos dois cenários, um para cada experimento, sendo eles:

- Criação de Cards

  - Abrir a página da aplicação
  - Aguardar o carregamento da página
  - Clicar em adicionar um novo Card
  - Clicar no botão de editar no Card criado
  - Aguardar o Modal de edição abrir
  - Editar o texto do título do Card com um novo valor
  - Clicar no botão de salvar edição no Modal
  - Clicar no botão de editar no Card criado
  - Aguardar o Modal de edição abrir
  - Editar o texto do subtítulo do Card com um novo valor
  - Clicar no botão de salvar edição no Modal
  - Clicar no botão de excluir no Card criado
  - Fechar página

- Mudança de Tema
  - Abrir a página da aplicação
  - Aguardar o carregamento da página
  - Aguarda o tema ser aplicado a tela
  - Clica no Select para ver as opções de temas
  - Clica no segundo tema da lista
  - Aguarda o tema ser aplicado a tela
  - Fechar página

Em cada experimento, foi medido o tempo de execução das ações dentro da aplicação e o consumo de memória que o código necessitou para realizar as devidas ações.

Todos os experimentos foram executados de forma automatizada utilizando o Puppeteer. Com a ferramenta, foram executados scripts que realizaram os cenários de teste. Os dados obtidos na execução dos scripts estão dispostos nas Tabela 7 e Tabela 8 que se encontram no Apêndice A - Dados coletados.

4.3 Resultados dos Testes

Após a coleta dos dados, usou-se da ferramenta Statdisk [25], a qual utiliza como base para seus algoritmos, as equações do livro [24]. Com o auxilio da ferramenta, foi possível realizar uma análise para cada experimento e obter resultados suficientes para as questões estatísticas.

4.3.1 Experimento - Criação de Cards

O experimento Criação de Cards, o qual a UI foi mostrada na Figura 9, irá simular a interação de um usuário com os elementos da tela, como o botão para criar novos cards, as entradas de texto para alterar o título e subtítulo e o botão para excluir um card criado.

Os dados utilizados para análise desse experimento, estão na Tabela 7, juntamente com o uso da ferramenta Statdisk, obtivemos os resultados a seguir.

4.3.1.1 Análise do Tempo de Execução

Análise do tempo de execução do experimento de Criação de Cards, teve seu início com o teste de normalidade dos dados usando *Ryan-Joiner test*, o resultado para os dados do Context API estão dispostos na Figura 11, e os dados do Redux na Figura 12, onde os dados atendem aos requisitos de normalidade.

Com a normalidade confirmada, usando a ferramenta Statdisk, obtive os dados dispostos na Tabela 3 e com eles segue a ánalise da seguinte hipótese:

- O tempo de execução da Context API é significativamente menor do que o tempo de execução do Redux.
  - **Hipótese Nula ($H_{0,\textbf{tempo}}$):**

$$H_{0,\text{tempo}} : \mu_{\text{Context API, tempo}} = \mu_{\text{Redux, tempo}}$$

  - **Hipótese Alternativa ($H_{a,\textbf{tempo}}$):**

$$H_{a,\text{tempo}} : \mu_{\text{Context API, tempo}} < \mu_{\text{Redux, tempo}}$$

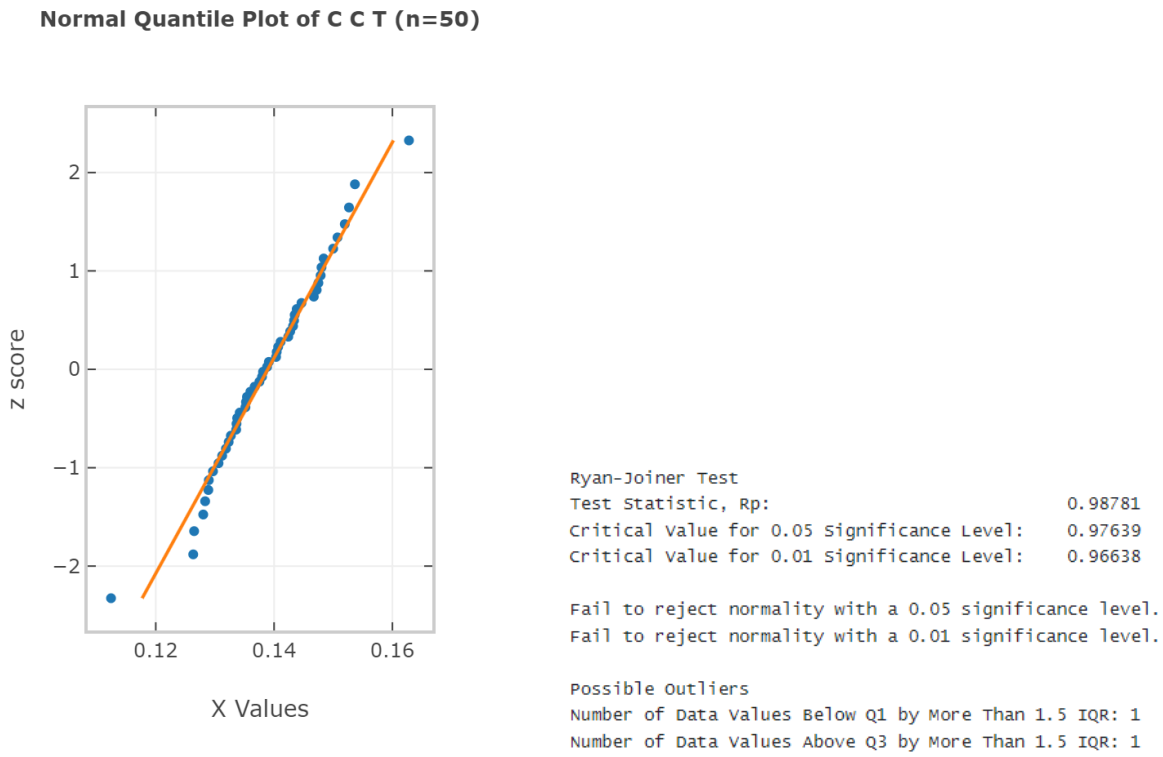


Figura 11: Teste de normalidade do Context API, Tempo de execução.
Fonte: .

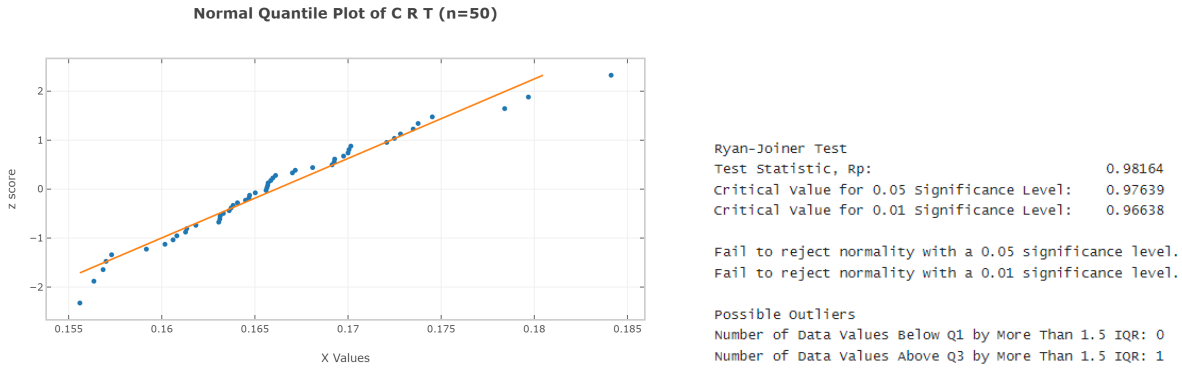


Figura 12: Teste de normalidade, Redux, Tempo de execução.
Fonte: .

O Valor-p obitido é menor que o nível de significância $\alpha = 0.01$, Há evidência suficiente para apoiar a afirmativa de que o tempo de execução da Context API é significativamente menor do que o tempo de execução do Redux.

Tabela 3: Resultados para o tempo de execução do experimento de Criação de Cards

Tempo de execução(s)	
μ_{Context}	0.13894
μ_{Redux}	0.16614
Test T	-17.749572
Intervalo de confiança	[-0.03083; -0.02357]
Valor-p	$1.12e^{-32}$

4.3.1.2 Análise do Consumo de Memória

Análise do consumo de memória do experimento de Criação de Cards, teve seu início com o teste de normalidade dos dados usando *Ryan-Joiner test*, o resultado para os dados do Context API estão dispostos na Figura 13, e os dados do Redux na Figura 14, onde os dados atendem aos requisitos de normalidade.

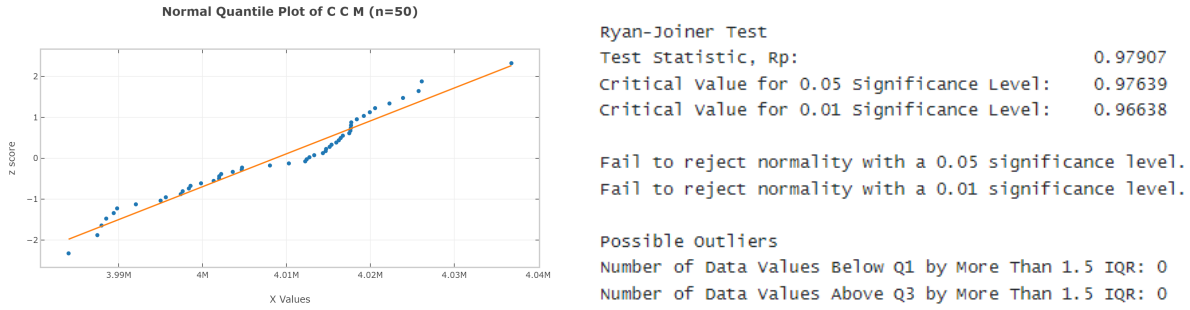


Figura 13: Teste de normalidade do Context API, Consumo de memória.
Fonte: .

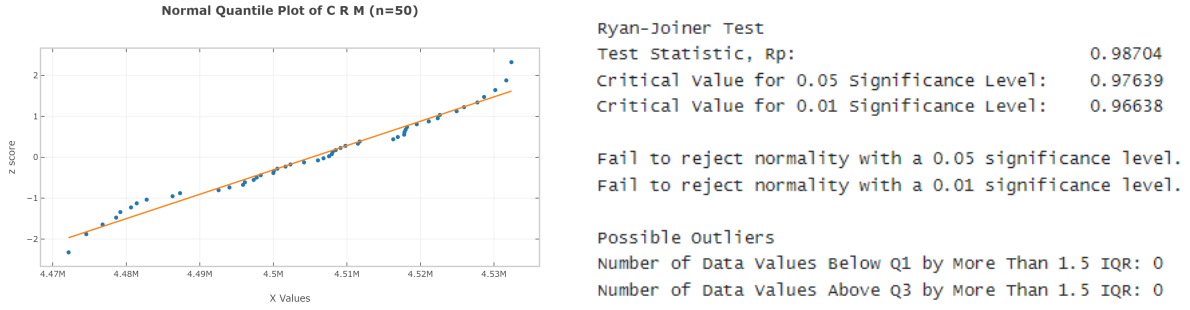


Figura 14: Teste de normalidade, Redux, Consumo de memória.
Fonte: .

Com a normalidade confirmada, usando a ferramenta Statdisk, obtive os dados dispostos na Tabela 4 e com eles segue a ánalise da seguinte hipótese:

Tabela 4: Resultados para o consumo de memória do experimento de Criação de Cards

Consumo de Memória(MB)	
μ_{Context}	3.823264
μ_{Redux}	4.296743
Test T	-171.42573
Intervalo de confiança	[-0.48002; -0.46701]
Valor-p	$2.69e^{-123}$

- O consumo de memória da Context API é significativamente menor do que o consumo de memória do Redux.

  – **Hipótese Nula ($H_{0,\textbf{memória}}$):**

  $$H_{0,\text{memória}} : \mu_{\text{Context API, memória}} = \mu_{\text{Redux, memória}}$$

  – **Hipótese Alternativa ($H_{a,\textbf{memória}}$):**

  $$H_{a,\text{memória}} : \mu_{\text{Context API, memória}} < \mu_{\text{Redux, memória}}$$

  O Valor-p obitido é menor que o nível de significância $\alpha = 0.01$, Há evidência suficiente para apoiar a afirmativa de que o consumo de memória da Context API é significativamente menor do que o consumo de memória do Redux.

4.3.2 Experimento - Mudança de Tema

O experimento Mudança de Tema, o qual a UI foi mostrada na Figura 10, irá simular a interação de um usuário com o seletor de tema, no qual o tema será mudado e com isso, todos os elementos em tela terão suas cores alteradas.

Os dados utilizados para análise desse experimento, estão na Tabela 8, juntamente com o uso da ferramenta Statdisk, obtivemos os resultados a seguir.

4.3.2.1 Análise do Tempo de Execução

Análise do tempo de execução do experimento de Mudança de Tema, teve seu início com o teste de normalidade dos dados usando *Ryan-Joiner test*, o resultado para

os dados do Context API estão dispostos na Figura 15, e os dados do Redux na Figura 16, onde os dados atendem aos requisitos de normalidade.

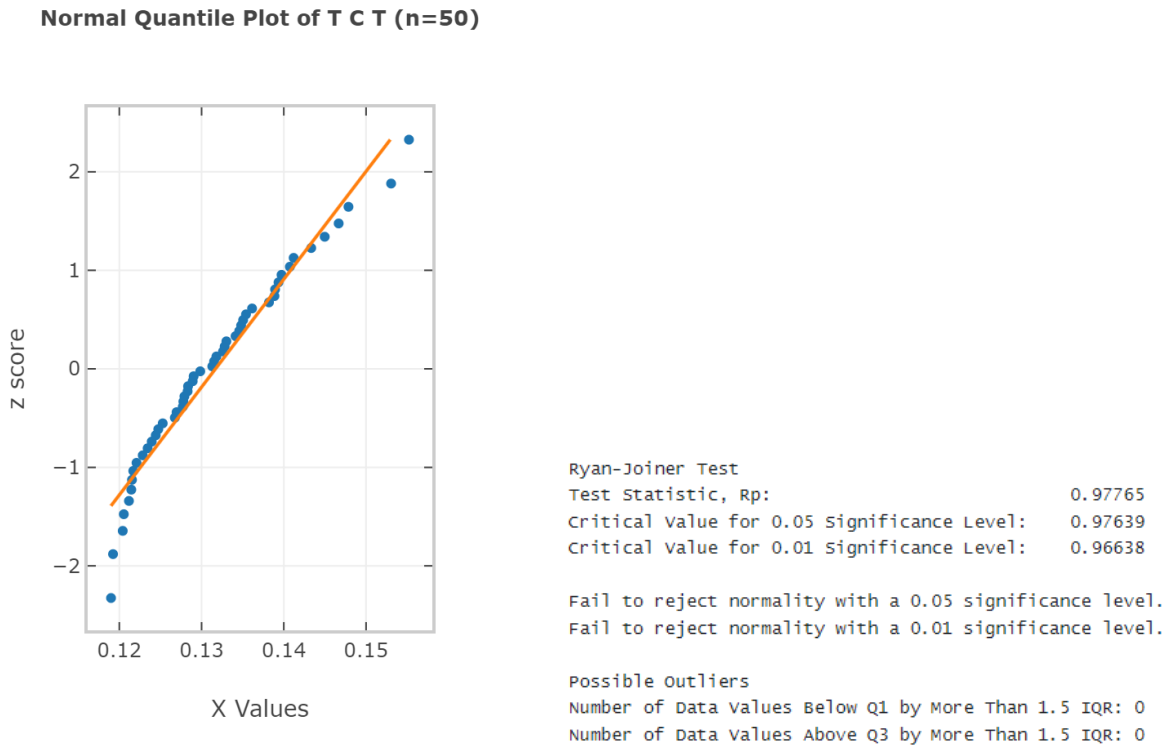


Figura 15: Teste de normalidade do Context API, Tempo de execução.
Fonte: .

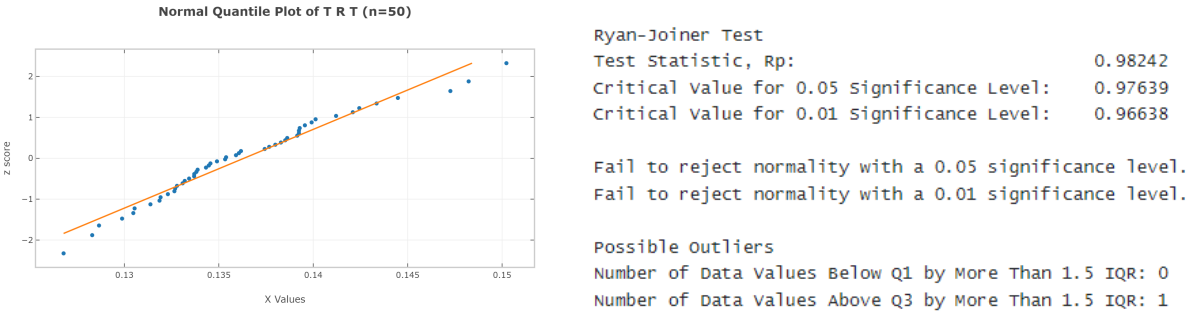


Figura 16: Teste de normalidade, Redux, Tempo de execução.
Fonte: .

Com a normalidade confirmada, usando a ferramenta Statdisk, obtive os dados dispostos na Tabela 5 e com eles segue a ánalise da seguinte hipótese:

- O tempo de execução da Context API é significativamente menor do que o tempo

de execução do Redux.

– **Hipótese Nula ($H_{0,\textbf{tempo}}$):**

$$H_{0,\text{tempo}} : \mu_{\text{Context API, tempo}} = \mu_{\text{Redux, tempo}}$$

– **Hipótese Alternativa ($H_{a,\textbf{tempo}}$):**

$$H_{a,\text{tempo}} : \mu_{\text{Context API, tempo}} < \mu_{\text{Redux, tempo}}$$

O Valor-p obitido é menor que o nível de significância $\alpha = 0.01$, Há evidência suficiente para apoiar a afirmativa de que o tempo de execução da Context API é significativamente menor do que o tempo de execução do Redux.

Tabela 5: Resultados para o tempo de execução do experimento de Mudança de Tema

Tempo de execução(s)	
μ_{Context}	0.131692
μ_{Redux}	0.136328
Test T	-3.197156
Intervalo de confiança	[-0.00808; -0.00119]
Valor-p	0.000934

4.3.2.2 Análise do Consumo de Memória

Análise do consumo de memória do experimento de Mudança de Tema, teve seu início com o teste de normalidade dos dados usando *Ryan-Joiner test*, o resultado para os dados do Context API estão dispostos na Figura 17, e os dados do Redux na Figura 18, onde os dados atendem aos requisitos de normalidade.

Com a normalidade confirmada, usando a ferramenta Statdisk, obtive os dados dispostos na Tabela 6 e com eles segue a ánalise da seguinte hipótese:

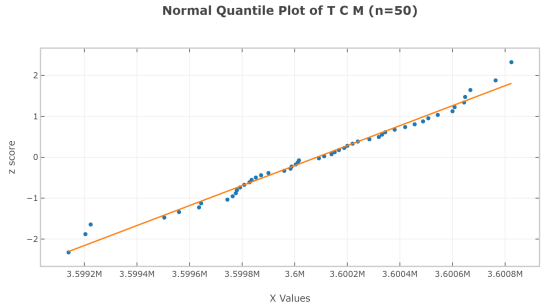
- O consumo de memória da Context API é significativamente menor do que o consumo de memória do Redux.

  – **Hipótese Nula ($H_{0,\textbf{memória}}$):**

$$H_{0,\text{memória}} : \mu_{\text{Context API, memória}} = \mu_{\text{Redux, memória}}$$

– **Hipótese Alternativa ($H_{a,\textbf{memória}}$):**

$$H_{a,\text{memória}} : \mu_{\text{Context API, memória}} < \mu_{\text{Redux, memória}}$$

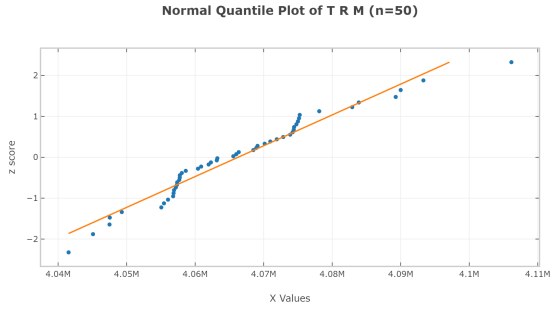



```
Ryan-Joiner Test
Test Statistic, Rp:                          0.99124
Critical Value for 0.05 Significance Level:  0.97639
Critical Value for 0.01 Significance Level:  0.96638

Fail to reject normality with a 0.05 significance level.
Fail to reject normality with a 0.01 significance level.

Possible Outliers
Number of Data Values Below Q1 by More Than 1.5 IQR: 0
Number of Data Values Above Q3 by More Than 1.5 IQR: 0
```

Figura 17: Teste de normalidade do Context API, Consumo de memória.
Fonte: .




```
Ryan-Joiner Test
Test Statistic, Rp:                          0.97749
Critical Value for 0.05 Significance Level:  0.97639
Critical Value for 0.01 Significance Level:  0.96638

Fail to reject normality with a 0.05 significance level.
Fail to reject normality with a 0.01 significance level.

Possible Outliers
Number of Data Values Below Q1 by More Than 1.5 IQR: 0
Number of Data Values Above Q3 by More Than 1.5 IQR: 1
```

Figura 18: Teste de normalidade, Redux, Consumo de memória.
Fonte: .

O Valor-p obitido é menor que o nível de significância $\alpha = 0.01$, Há evidência suficiente para apoiar a afirmativa de que o consumo de memória da Context API é significativamente menor do que o consumo de memória do Redux.

Tabela 6: Resultados para o consumo de memória do experimento de Mudança de Tema

Consumo de Memória(MB)	
μ_{Context}	3.43267
μ_{Redux}	3.87754
Test T	-254.64557
Intervalo de confiança	[-0.44733; - 0.44166]
Valor-p	$2.29e^{-140}$

4.4 Discussão

Os resultados indicaram que a Context API tende a ter um desempenho melhor em comparação ao Redux, especialmente em aplicações mais simples. Nos experimentos realizados, a Context API não só demonstrou uma maior eficiência em termos de velocidade de execução e consumo de memória, mas também se mostrou mais fácil de desenvolver, proporcionando uma vantagem adicional para projetos menores e menos complexos.

No entanto, deve-se considerar a complexidade das aplicações utilizadas nos experimentos. Embora a Context API tenha se destacado nas aplicações testadas, a situação pode ser diferente em projetos mais complexos. Em tais casos, mesmo que a Context API mantenha sua eficiência no consumo de recursos, o Redux oferece uma estrutura de código mais legível e organizada. Essa característica é particularmente benéfica quando se trata de gerenciar uma grande quantidade de estados, facilitando a manutenção e escalabilidade do código.

Portanto, a escolha entre Redux e Context API deve levar em conta o tamanho e a complexidade da aplicação. Para projetos menores e menos complexos, a Context API pode ser a melhor opção devido à sua simplicidade e eficiência. Já em projetos maiores, a estrutura mais robusta do Redux pode proporcionar uma melhor organização e gerenciamento do estado da aplicação.

4.5 Conclusão dos Resultados Experimentais

Com a utilização do Puppeteer para executar os experimentos e coletar dados suficientes para as devidas aplicações estatísticas, chegou-se ao resultado de que em aplicações mais simples como as aqui utilizadas, a Context API apresenta uma clara vantagem em ser utilizada devido a seu melhor consumo de recursos. Porém vale ressaltar a importância de considerar outros fatores em quanto a escolha de qual ferramenta utilizar para gerenciar os estados de uma aplicação real, pois conforme a complexidade aumente, o Redux pode trazer pontos vantajosos em relação a Context API, que o tornem uma opção mais vível.

5 CONCLUSÃO E TRABALHOS FUTUROS

A análise comparativa entre Redux e Context API demonstrou que, para aplicações menores, a Context API apresenta um desempenho superior em termos de velocidade de execução e consumo de memória. No entanto, à medida que a complexidade da aplicação aumenta, Redux pode oferecer vantagens em termos de escalabilidade e organização do código, um resultado similar ao encontrado por Le(2021) em seu artigo *'Comparison of State Management Solutions between Context API and Redux Hook in ReactJS'* [26].

Os resultados sugerem que desenvolvedores devem considerar o tamanho e a complexidade da aplicação ao escolher entre Redux e Context API. Para projetos menores ou protótipos rápidos, a Context API pode ser a melhor opção devido aos seus benefícios e estrutura. Por outro lado, para aplicações maiores e mais complexas, Redux pode proporcionar uma melhor organização e manutenção do código.

Este estudo foi realizado com base em cenários de teste específicos e pode não abranger todas as situações possíveis no desenvolvimento de aplicações React. Fatores como a regra de negócios da aplicação e a interação com outras bibliotecas também podem influenciar o desempenho das ferramentas no gerenciamento de estado.

Para futuras pesquisas, seria interessante explorar o desempenho de Redux e Context API em aplicações mais robustas e analisar o comportamento de escalabidade conforme uma aplicação se expande. Além disso, a investigação de outras ferramentas de gerenciamento de estado pode proporcionar uma visão mais abrangente das opções disponíveis para desenvolvedores, como por exemplo o MobX [27].

Em conclusão, tanto Redux quanto Context API têm seus méritos e o melhor uso de cada um depende do contexto específico da aplicação. Este estudo fornece uma base para desenvolvedores tomarem decisões informadas sobre a escolha de ferramentas de gerenciamento de estado em React, contribuindo para a eficiência e qualidade do desenvolvimento de software.

REFERÊNCIAS

[1] Stack Overflow. *Developer Survey 2023*. Available at: <https://survey.stackoverflow.co/2023/most-popular-technologies-language-prof>. Accessed in: 19/02/2024.

[2] Michele Lopes. *React: o que é e como funciona*. Available at: <https://ebaconline.com.br/blog/react-o-que-e-como-funciona>. Accessed in: 18/06/2024.

[3] Stephen Arancio. *ReactJS: A brief history*. Available at: <https://medium.com/@sjarancio/reactjs-a-brief-history-3c1e969a477f>. Accessed in: 19/02/2024.

[4] Dan Abramov and the Redux documentation. *A (Brief) History of Redux*. Available at: <https://redux.js.org/understanding/history-and-design/history-of-redux>. Accessed in: 19/02/2024.

[5] Brian Vaughn. *React v16.3.0: Novos ciclos de vida e API de contexto*. Available at: <https://pt-br.legacy.reactjs.org/blog/2018/03/29/react-v-16-3.html>. Accessed in: 19/02/2024.

[6] Versha Gupta. *React state management: What is it and why to use it?* Available at: <https://www.loginradius.com/blog/engineering/react-state-management/>. Accessed in: 19/02/2024.

[7] React documentation. *Construindo a UI*. Available at: <https://pt-br.react.dev/learn/describing-the-ui>. Accessed in: 19/02/2024.

[8] tutorialspoint. *ReactJS - Overview*. Available at: <https://www.tutorialspoint.com/reactjs/reactjs$_o$*verview.htm* : : *text* = *React*>.*Accessedin* : 19/02/2024.

[9] Sharvin Shah. *How to Use Flux to Manage State in ReactJS - Explained with an Example*. Available at: <https://www.freecodecamp.org/news/how-to-use-flux-in-react-example/>. Accessed in: 19/02/2024.

[10] Durga Prasad Acharya. *Mais de 15 Tutoriais e Recursos de React para Desenvolvedores*. Available at: <https://kinsta.com/pt/blog/tutoriais-e-recursos-react/>. Accessed in: 21/02/2024.

[11] React documentation. *Introduzindo JSX*. Available at: <https://pt-br.legacy.reactjs.org/docs/introducing-jsx.html>. Accessed in: 21/02/2024.

[12] React documentation. *Writing Markup with JSX*. Available at: <https://pt-br.react.dev/learn/writing-markup-with-jsx>. Accessed in: 21/02/2024.

[13] Especialista Coodesh. *O que é camelCase?* Available at: <https://coodesh.com/blog/dicionario/o-que-e-camelcase/>. Accessed in: 27/06/2024.

[14] Patrick Porto. *Escalando o gerenciamento de estados no React*. Available at: <https://medium.com/loftbr/escalando-o-gerenciamento-de-estados-no-react-7704dab2d56d>. Accessed in: 21/02/2024.

[15] Dan Abramov and the Redux documentation. *Getting Started with Redux*. Available at: <https://redux.js.org/introduction/getting-started>. Accessed in: 23/02/2024.

[16] tutorialspoint. *Redux - Data Flow*. Available at: <https://www.tutorialspoint.com/redux/redux$_{d}ata_{f}low.htm$>.$Accessedin$: 23/02/2024.

[17] Pablo Henrique Aguiar Cavalcante. *Redux: Um tutorial prático e simples!* Available at: <https://blog.geekhunter.com.br/redux-um-tutorial-pratico-e-simples/>. Accessed in: 23/02/2024.

[18] React documentation. *Context*. Available at: <https://pt-br.legacy.reactjs.org/docs/context.html>. Accessed in: 23/02/2024.

[19] Dickson Boateng. *How to Use the React Context API in Your Projects*. Available at: <https://www.freecodecamp.org/news/context-api-in-react/>. Accessed in: 23/02/2024.

[20] Puppeteer documentation. *Puppeteer*. Available at: <https://pptr.dev/>. Accessed in: 01/03/2024.

[21] Google. *DevTools*. Available at: <https://developer.chrome.com/docs/devtools?hl=pt-br>. Accessed in: 27/02/2024.

[22] Kayce Basques. *Analisar o desempenho do ambiente de execução*. Available at: <https://developer.chrome.com/docs/devtools/performance?hl=pt-br>. Accessed in: 27/02/2024.

[23] Puppeteer documentation. *Puppeteer*. Available at: <https://pptr.dev/api/puppeteer.page.metrics>. Accessed in: 01/03/2024.

[24] TRIOLA, M. F. *Introducao a Estatistica*. 12. ed. [S.l.]: LTC, 2017.

[25] Triola Stats. *Statdisk*. Available at: <https://www.statdisk.com/>. Accessed in: 31/05/2024.

[26] LE, T. Comparison of state management solutions between context api and redux hook in reactjs. 2021.

[27] Mobx Documentation. *MobX*. Available at: <https://mobx.js.org/react-integration.html>. Accessed in: 02/06/2024.

Apêndice A - Dados coletados

Tabela 7: Dados coletados com o Puppeteer - Aplicação: Criação de Cards

Tempo de execução - Context API(s)	**Tempo de execução - Redux(s)**	**Memória - Context API(MB)**	**Memória - Redux(MB)**
0.142403	0.163305	3.812500	4.290863
0.150722	0.161342	3.825195	4.293945
0.135255	0.170001	3.802734	4.270706
0.147496	0.184127	3.803711	4.291992
0.14999	0.16383	3.811035	4.316528
0.129682	0.161276	3.840820	4.316406
0.134197	0.173501	3.820312	4.319336
0.130614	0.160809	3.832031	4.271484
0.139119	0.172491	3.815430	4.316406
0.132702	0.165643	3.815918	4.295776
0.143212	0.169142	3.832031	4.270020
0.12803	0.160172	3.828125	4.319336
0.131866	0.165965	3.805420	4.266113
0.140445	0.15685	3.815918	4.269531
0.141106	0.172817	3.818359	4.291748
0.133595	0.167015	3.815430	4.319336
0.151955	0.157306	3.829590	4.290283
0.135998	0.163723	3.832031	4.318848
0.136722	0.163116	3.814697	4.266663
0.140338	0.161832	3.817871	4.309875
0.135158	0.163134	3.841309	4.295410
0.135426	0.163059	3.820801	4.287354
0.147203	0.165599	3.828125	4.265869
0.148019	0.179687	3.826172	4.274902

Tempo de execução - Context API(s)	**Tempo de execução - Redux(s)**	**Memória - Context API(MB)**	**Memória - Redux(MB)**
0.147874	0.178411	3.838379	4.261230
0.148383	0.169284	3.835938	4.319824
0.137968	0.1681	3.825195	4.313721
0.144635	0.165685	3.811035	4.290527
0.126491	0.16571	3.839355	4.295410
0.153668	0.16611	3.803833	4.287354
0.137525	0.174525	3.818359	4.289978
0.128349	0.164064	3.808105	4.319580
0.128891	0.157005	3.832031	4.291992
0.138822	0.165854	3.835938	4.292480
0.133765	0.160606	3.832031	4.295776
0.152664	0.170153	3.811035	4.314697
0.142723	0.170052	3.828125	4.295532
0.146717	0.164494	3.814453	4.317139
0.138136	0.172078	3.832031	4.285706
0.112445	0.164715	3.844238	4.295532
0.131229	0.165019	3.818359	4.291748
0.133652	0.173764	3.805664	4.309082
0.126337	0.167168	3.804688	4.285156
0.143486	0.169761	3.832031	4.296631
0.162802	0.155602	3.803711	4.263611
0.140693	0.159173	3.861816	4.313721
0.132329	0.164684	3.811035	4.283936
0.143378	0.169255	3.828125	4.309570
0.12897	0.156355	3.832031	4.315308
0.14383	0.16362	3.832031	4.313721

Tabela 8: Dados coletados com o Puppeteer - Aplicação: Mudança de Tema

Tempo de execução - Context API(s)	**Tempo de execução - Redux(s)**	**Memória - Context API(MB)**	**Memória - Redux(MB)**
0.131803	0.150228	3.429688	3.888672
0.12771	0.142095	3.429688	3.888092
0.123936	0.138287	3.429688	3.874023
0.128355	0.136172	3.429688	3.875488
0.153049	0.133691	3.429932	3.913696
0.118987	0.142437	3.429688	3.875488
0.124424	0.133812	3.429688	3.895508
0.13152	0.136063	3.429688	3.876343
0.138937	0.138621	3.429932	3.875732
0.126747	0.132689	3.429688	3.864746
0.121535	0.128295	3.429932	3.888428
0.122087	0.133195	3.429688	3.908325
0.126958	0.13054	3.429688	3.875610
0.12041	0.133877	3.429688	3.868408
0.121693	0.134556	3.429932	3.868530
0.138864	0.132302	3.429688	3.888672
0.132605	0.132648	3.429688	3.875122
0.129839	0.137992	3.429688	3.880432
0.132807	0.140124	3.429688	3.880371
0.128918	0.137668	3.429688	3.875610
0.134808	0.12678	3.429688	3.892944
0.1354	0.135334	3.429688	3.855530
0.134132	0.134471	3.429932	3.887939
0.125272	0.137429	3.429688	3.888306
0.124735	0.1337	3.429688	3.887207
0.121452	0.132785	3.429688	3.880371
0.127785	0.133426	3.429688	3.875610

Tempo de execução - Context API(s)	**Tempo de execução - Redux(s)**	**Memória - Context API(MB)**	**Memória - Redux(MB)**
0.140735	0.139242	3.429688	3.868286
0.14667	0.138517	3.429932	3.875610
0.13614	0.143353	3.429688	3.855103
0.135056	0.141208	3.429688	3.875305
0.122843	0.129878	3.429688	3.858154
0.147865	0.13432	3.429688	3.880859
0.119232	0.147261	3.429688	3.864624
0.131304	0.128656	3.429688	3.875488
0.141196	0.13924	3.429688	3.880859
0.121166	0.144482	3.429688	3.869385
0.143332	0.131916	3.429688	3.895264
0.13458	0.130472	3.429688	3.888458
0.123438	0.139283	3.429688	3.875244
0.138197	0.133083	3.429688	3.880188
0.139356	0.139557	3.429688	3.888672
0.144975	0.139152	3.429688	3.888092
0.139721	0.135385	3.429688	3.875488
0.133014	0.131379	3.429688	3.894897
0.128299	0.148229	3.429688	3.887573
0.129034	0.131859	3.429688	3.880737
0.127911	0.139915	3.429688	3.864807
0.15522	0.135908	3.429688	3.870117
0.120541	0.134901	3.429688	3.880066