Análise de desempenho de controladores de estado do framework Flutter *

Igor Dal Cero Rocha† Jorge Luis Boeira Bavaresco‡

2024

Resumo

Este trabalho apresenta uma análise comparativa do desempenho de diferentes controladores de estado no *framework "Flutter"*: *"setState"*, *"Provider"*, *"BLoC"* e *"MobX"*. O gerenciamento de estado é um elemento crucial no desenvolvimento de aplicativos móveis, especialmente em cenários que exigem atualizações frequentes e responsivas da interface do usuário. A pesquisa foi motivada pela questão: "Qual controlador de estado oferece o melhor desempenho para uma aplicação *"Flutter"*?", e teve como objetivo avaliar o impacto de cada abordagem no consumo de memória, uso de CPU e tempo de resposta. Para isso, foram desenvolvidos aplicativos de teste que implementaram as funcionalidades de navegação e atualização de estado utilizando cada controlador. O desempenho foi medido utilizando o Snapdragon Profiler, analisando métricas como consumo de memória, uso de CPU e tempo de execução. Os resultados revelaram que o *"setState"* é mais adequado para funcionalidades simples, enquanto o *"Provider"* mostrou-se eficiente em aplicações que exigem atualizações frequentes. O *"BLoC"*, por sua vez, destacou-se em cenários de alta complexidade, devido à sua capacidade de separar a lógica de negócios da interface. Já o *"MobX"* demonstrou excelente desempenho em aplicações que demandam alta reatividade, apesar de seu maior consumo de recursos em cenários intensivos. A análise comparativa contribui para a compreensão das vantagens e limitações de cada controlador, auxiliando desenvolvedores na escolha da abordagem mais adequada para diferentes tipos de aplicações. Este estudo oferece uma base teórica e prática valiosa, visando otimizar o desempenho e a experiência do usuário em projetos desenvolvidos com *"Flutter"*.

Palavras-chaves: Flutter. Gerenciamento de estado. Análise de desempenho.

*Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, na cidade de Passo Fundo, em (2024).

†Dados do autor.

‡Orientador do trabalho (dados).

1 Introdução

Nos últimos anos, o framework *"Flutter"* tem se destacado como uma ferramenta poderosa para o desenvolvimento de aplicativos móveis multiplataforma, atraindo desenvolvedores e empresas devido à sua capacidade de criar interfaces ricas e consistentes para *Android* e *iOS* (TASHILDAR et al., 2020). Contudo, a escolha do controlador de estado adequado para as aplicações representa um desafio significativo. O gerenciamento de estado é crucial para o desempenho e a usabilidade das aplicações, especialmente em projetos mais complexos, onde a atualização eficiente da interface do usuário é essencial. Este estudo teve como tema principal a análise do framework *"Flutter"* e de seus controladores de estado, com o objetivo de avaliar seu desempenho.

A problemática de pesquisa que motiva este trabalho é a seguinte: "Qual controlador de estado oferece o melhor desempenho para a aplicação?". Essa dúvida é relevante devido às variações de desempenho que cada controlador pode oferecer, uma vez que são construídos de maneiras distintas, resultando em diferentes níveis de consumo de CPU, memória e tempo de execução. Trabalhar com este tema é fundamental não apenas para compreender as nuances do desenvolvimento com *"Flutter"*, mas também para fornecer um recurso valioso para desenvolvedores e profissionais de tecnologia que buscam otimizar suas aplicações.

A relevância deste trabalho se justifica por duas razões principais. Teoricamente, ele contribui para o avanço do conhecimento na área de gerenciamento de estado em aplicações móveis, um tópico que, apesar de sua importância, ainda carece de estudos aprofundados. Praticamente, oferece insights que podem guiar a escolha do controlador de estado mais apropriado, melhorando a eficiência e a experiência do usuário nas aplicações desenvolvidas.

Este documento está estruturado de forma a apresentar inicialmente os conceitos fundamentais e o referencial teórico relacionado ao *"Flutter"* e seus controladores de estado. Em seguida, a metodologia aplicada detalha o desenvolvimento de aplicativos de teste utilizando diferentes controladores. Os resultados e análises são apresentados de forma comparativa, destacando o impacto de cada abordagem em métricas de desempenho. Por fim, as considerações finais discutem os aprendizado do estudo,desafios encontrados e direções futuras para o aprofundamento da pesquisa na área.

2 Referencial teórico

Nesta seção, será abordada o referencial teórico fundamental para a compreensão e aplicação de técnicas de gerenciamento de estado no desenvolvimento de aplicativos com *"Flutter"*. *"Flutter"*, um *framework* de código aberto criado pelo Google, revolucionou o desenvolvimento de aplicativos multiplataforma com sua abordagem única e eficiente. Para lidar com a complexidade crescente das aplicações, especialmente aquelas que exigem uma interação intensiva do usuário, o gerenciamento de estado se torna um aspecto crucial. Examinaremos as principais técnicas e ferramentas utilizadas para gerenciar o estado em *"Flutter"*, incluindo o uso de *"BLoC"*, *"Provider"*, *"MobX"* e o controlador nativo *"setState"*. Cada uma dessas abordagens oferece vantagens

específicas e é adequada para diferentes tipos de aplicativos e necessidades de desenvolvimento. Vamos explorar essas ferramentas e técnicas em detalhes para entender suas funcionalidades e aplicações práticas.

2.1 Flutter

"Flutter" é um *framework* de código aberto utilizada para o desenvolvimento de aplicativos móveis, web e desktop. Este *framework* foi desenvolvido pelo Google e lançado no ano de 2017. *"Flutter"* foi desenvolvido utilizando a linguagem Dart, que é uma linguagem criada e otimizada pelo Google com o objetivo de ser utilizada no desenvolvimento de aplicativos móveis e web (ZULISTIYAN; ADRIAN; WIBOWO, 2024).

Dart é uma linguagem versátil que vai além de aplicações móveis. Ele possui a capacidade de compilar para JavaScript, o que o torna possível de ser utilizado para o desenvolvimento web. Isso possibilita que os desenvolvedores usem uma única linguagem de programação com o mesmo código base para construir aplicações multiplataformas, incluindo iOS, Android e web, reduzindo tempo e esforço no período de desenvolvimento. O design do Dart enfatiza a produtividade e facilita o aprendizado. Sua sintaxe é limpa e familiar para desenvolvedores com experiência em linguagens orientadas a objetos, como Java ou C#. Isso diminui a curva de aprendizado, tornando-o mais acessível para uma ampla gama de desenvolvedores (DART, 2024).

2.2 Controladores de Estado em Flutter

Controladores de estado é uma técnica utilizada para armazenar os estados em uma aplicação, sendo especialmente importantes no desenvolvimento de aplicativos móveis. O estado de uma aplicação refere-se á representação de informações que podem mudar ao longo do tempo, como a seleção de um item em uma lista, a autenticação de um usuário ou o conteúdo exibido na interface do usuário. Um gerenciamento eficaz do estado é essencial para garantir que essas mudanças sejam refletidas de maneira eficiente e responsiva.

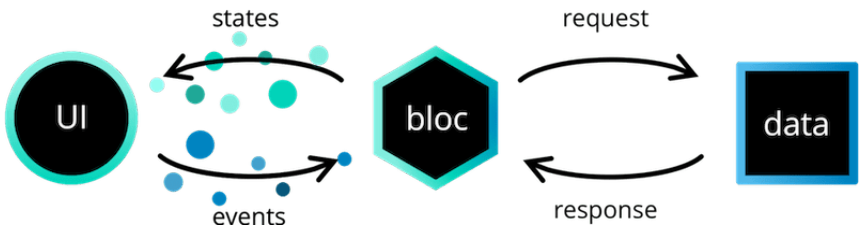
No *"Flutter"*, os controladores de estado desempenham um papel crucial, particularmente em aplicativos complexos exigem uma interação intensiva entre o usuário e a aplicação. Algumas das técnicas de gerenciamento de estado mais utilizados incluem *"setState"*, *"Provider"*, *"BLoC"* e *"MobX"* (ZULISTIYAN; ADRIAN; WIBOWO, 2024).

2.3 BLoC (Business Logic Component)

O *"BLoC"* é uma biblioteca usada no desenvolvimento com o *"Flutter"* para separar a lógica de negócios da interface do usuário. Seu principal propósito é aumentar a modularidade, testabilidade e reutilização do código. O *"BLoC"* funciona com base no conceito de eventos e estados. Os eventos representam interações do usuário ou mudanças de dados, que são processados pela lógica de negócios no *"BLoC"*. Os estados resultantes alteram os *widgets* de interface do usuário, como demonstrado na Figura 1 (ANGELOV., 2023).

Na versão mais recente do *"BLoC"*, o gerenciamento de estados e eventos foi simplificado, eliminando a complexa dependência de *Stream* e *Sink*. Agora, o *"BLoC"* facilita o processo de eventos e entrega novos estados de maneira integrada com o *"Flutter"*. Isso torna a separação entre UI e lógica de negócios mais intuitiva, acelerando o processo de desenvolvimento e simplificando os testes e a manutenção do código. (ZULISTIYAN; ADRIAN; WIBOWO, 2024).

Figura 1 – Diagrama de Fluxo BLoC, ilustrando o gerenciamento de estado e eventos



Fonte: (LTD., 2023)

2.4 Provider

O *"Provider"* é um conjunto de classes projetadas para simplificar o gerenciamento de estado no *"Flutter"*. Ele encapsula o conceito do *Inherited Widget*, tornando seu uso mais simples e reutilizável. A equipe de desenvolvimento do *"Flutter"* recomenda o *"Provider"* como o método preferencial para gerenciar o estado em aplicativos.(Flutter, 2023). O *"Provider"* passa informações de mudança de estado através de seus componentes. Quando ocorre uma alteração nos dados na classe de gerenciamento, ele notifica os observadores (ou seja, os *widgets* dependentes) para atualizarem sua interface de usuário. O fluxo básico do *"Provider"* envolve os seguintes passos. Inicialmente, é criado um modelo que representa o estado que será gerenciado. Isso pode ser qualquer coisa, desde dados de usuários até configurações do aplicativo. Logo após isso, será criada uma instância do *"Provider"*, passando o modelo como argumento. O *"Provider"* manterá o estado e fornecerá acesso a ele para os *widgets* filhos.

Os *widgets* que terão acesso ao estado serão registrados como consumidores pelo *"Provider"*. Isso será feito utilizando o *widget 'Consumer'* ou o método *'context.watch<Model>()'*. Quando ocorre alguma alteração no estado (por exemplo, quando um usuário faz login), o *"Provider"* notifica os consumidores e atualiza a interface do usuário automaticamente (RABELO, 2020).

2.5 MobX Biblioteca de Gerenciamento de estado

O *"Mobx"* é uma biblioteca de gerenciamento de estado que simplifica a conexão dos dados reativos do aplicativo coma interface do usuário. Ele foi baseado em observáveis e reações, permitindo que o *widgets* sejam automaticamente atualizados quando os dados de observação são modificados. Isso é particularmente útil para aplicativos que exigem atualizações em tempo

real (MOZ, 2020). O *"Mobx"* funciona com base em três conceitos principais: observáveis, ações e reações:

- Observáveis: São as propriedades que irão ser monitoradas pelo *"MobX"*, para essas bibliotecas sejam observáveis pela aplicação é necessário que cada propriedade esteja com o decorador *"@observable"*.
- Ações: São funções que modificam os observáveis. No *"Mobx"*, qualquer função que modifica os observáveis é marcada com um decorador *"@action"*.
- Reações: São os efeitos colaterais que ocorrem em responta á mudança nos observáveis. No *"Mobx"*, as reações são criadas usando funções como *"autorun"*, *"when"*, *"reaction"*.

Quando um observável muda, todas as reações que dependem desse observável são automaticamente disparadas. Isso garante que a interface do usuário esta sempre sincronizada com o estado (SANTOS, 2020).

2.6 SetState

O controlador *"setState"* é a ferramenta nativa do *"Flutter"* para o gerenciamento de estado em *widgets*. Seu principal propósito é atualizar o estado interno do *widget* e, consequentemente, reconstruir a interface do usuário para refletir as mudanças. O controlador "setState" é utilizado por *widgets* do tipo *StatefulWidget*, que são compostos por duas classes principais: o StatefulWidget, que define a estrutura do *widget* e cria a instância da classe *State*, onde o estado do *widget* é gerenciado.

A classe *"State"* é onde o estado de cada *widget* é armazenado e gerenciado. Esta classe é mutável e contém a lógica necessária para atualizar o estado e reconstruir a interface do usuário. A função *"setState"* é usada dentro da classe *'State'* para atualizar o estado do *widget*. Quando chamada, ela executa a função fornecida e, em seguida, chama o método *"build"* para reconstruir o *widget* com o novo estado.

Embora a função *"setState"* seja uma solução nativa e poderosa para o gerenciamento de estado em *"Flutter"*, ela é mais adequada para estados simples e localizados. Para estados mais complexos ou que precisam ser compartilhados entre vários *widgets*, outros controladores de estado como *"Provider"*, *"BLoC"* ou *"MobX"* podem ser mais apropriados (ALMEIDA, 2024).

2.7 Trabalhos Relacionados

No desenvolvimento de aplicativos *"Flutter"*, a escolha do controlador de estado é fundamental para garantir a eficiência e o desempenho das aplicações. Estudos recentes destacam a importância das bibliotecas de gerenciamento de estado, que permitem controlar a atualização dos *widgets* com base em mudanças de estado, evitando o reconstrução desnecessária de componentes de interface. Este trabalho se baseia na análises de desempenho de diferentes abordagens de

gerenciamento de estado, em especial as bibliotecas *"BLoC"*, *"GetX"* e *"Provider"*, para auxiliar na seleção da biblioteca mais eficiente de acordo com as necessidades de cada aplicação.

O estudo de Zulistiyan, Adrian e Wibowo (2024) investiga o desempenho das bibliotecas *"BLoC"* e *"GetX"* em *"Flutter"* focando no uso da CPU e memória. A pesquisa avalia o comportamento deste controladores com diferentes tamanhos de dados (1.000, 5000 e 10000 entradas) o que revela que o controlador *"GetX"* é mais eficiente na utilização de memória em conjuntos de dados menores, enquanto o *"BLoC"* se mostra mais estável e eficiente em CPU em cargas de dados maiores. Essas descobertas fornecem orientações importantes para desenvolvedores ao escolher o controlador ideal para diferentes cenários, sendo *"GetX"* o mais recomendado para aplicações leves e rápidas, e *"BLoC"* é recomendado para aplicações que exigem uma maior robustez e escalabilidade (ZULISTIYAN; ADRIAN; WIBOWO, 2024).

Outro estudo relevante, realizado por Prayoga et al. (2021) , compara o desempenho dos controladores *"BLoC"* e *"Provider"* em *"Flutter"*, com ênfase na utilização de CPU, uso de memória e tempo de execução. Os resultados do estudo indicam que tanto o *"BLoC"* quando *"Provider"* superam o método padrão de *"Flutter"*, o *"setState"*, em eficiência. Para 1000 e 10000 entradas, *"BLoC"* e *"Provider"* mostram um melhor desempenho, com um menor consumo de recursos e tempo de execução reduzidos. Esse estudo destaca que o uso do *"BLoC"* e *"Provider"* reduz o número de *widgets* reconstruídos, melhorando a performance da aplicação, especialmente em widgets de nível inferior, onde as mudanças locais ocorrem com maior frequência (PRAYOGA et al., 2021).

Esses trabalhos são fundamentais para o presente estudo, pois oferecem uma base de comparação e aprofundam o entendimento sobre as vantagens e limitações de cada biblioteca de estado no *"Flutter"*. Ao analisar esses resultados, este trabalho visa contribuir com uma avaliação adicional, incluindo o controlador *"MobX"*, para oferecer uma visão abrangente das opções de gerenciamento de estado e sua aplicabilidade em diferentes tipos de aplicações desenvolvidas com *"Flutter"*.

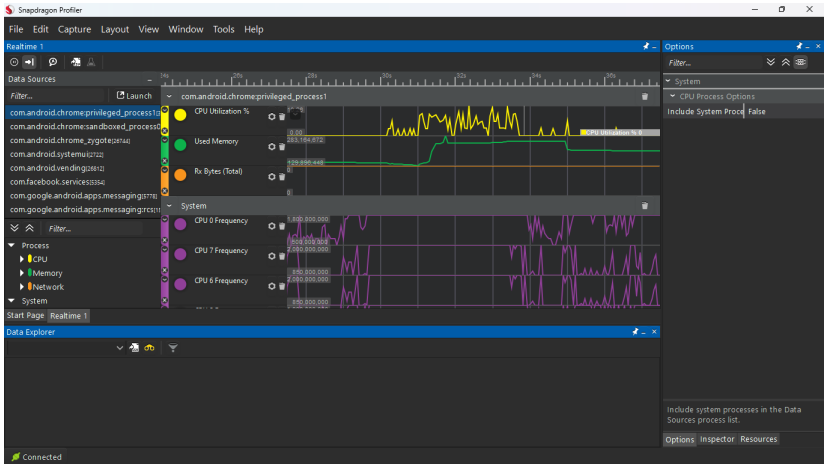
2.8 Snapdragon Profiler

O *"Snapdragon Profiler"* é uma ferramenta desenvolvida pela Qualcomm para análise e otimização de desempenho em dispositivos com processadores Snapdragon, conforme ilustrado na Figura 2. Ele oferece métricas detalhadas, como uso de CPU, memória, consumo de energia e desempenho gráfico, permitindo identificar gargalos e otimizar aplicativos móveis. Na imagem, observa-se a interface do *"Snapdragon Profiler"* exibindo gráficos que detalham a utilização da CPU, frequência dos núcleos e consumo de memória de um processo específico. Essa ferramenta é amplamente utilizada no desenvolvimento e teste de aplicações, ajudando a compreender como os recursos de hardware estão sendo utilizados e como diferentes escolhas de implementação impactam a eficiência geral da aplicação.

No contexto do estudo sobre controladores de estado no *"Flutter"*, o *"Snapdragon Profiler"* foi utilizado para capturar métricas como consumo de CPU, memória e tempo de resposta em aplicativos de teste que implementaram diferentes abordagens de gerenciamento de estado. Essa

análise detalhada permitiu comparar o impacto de cada controlador, fornecendo informações práticas para otimizar o desempenho de aplicativos móveis (QUALCOMM, 2024).

Figura 2 – Ferramenta Snapdragon Profile



Fonte: Própria

3 Metodologia

Nesta seção, será apresentada a metodologia adotada para analisar o desempenho dos controladores de estado no *framework "Flutter"*. O foco será especificamente nos controladores *"setState"*,*"BLoC"*, *"Provider"* e *"MobX"*, com o objetivo de avaliar e comparar seu impacto na eficiência e no desempenho dos aplicativos *"Flutter"*. Para isso, será utilizado o Snapdragon Profiler da Qualcomm para coletar e analisar as métricas relevantes.

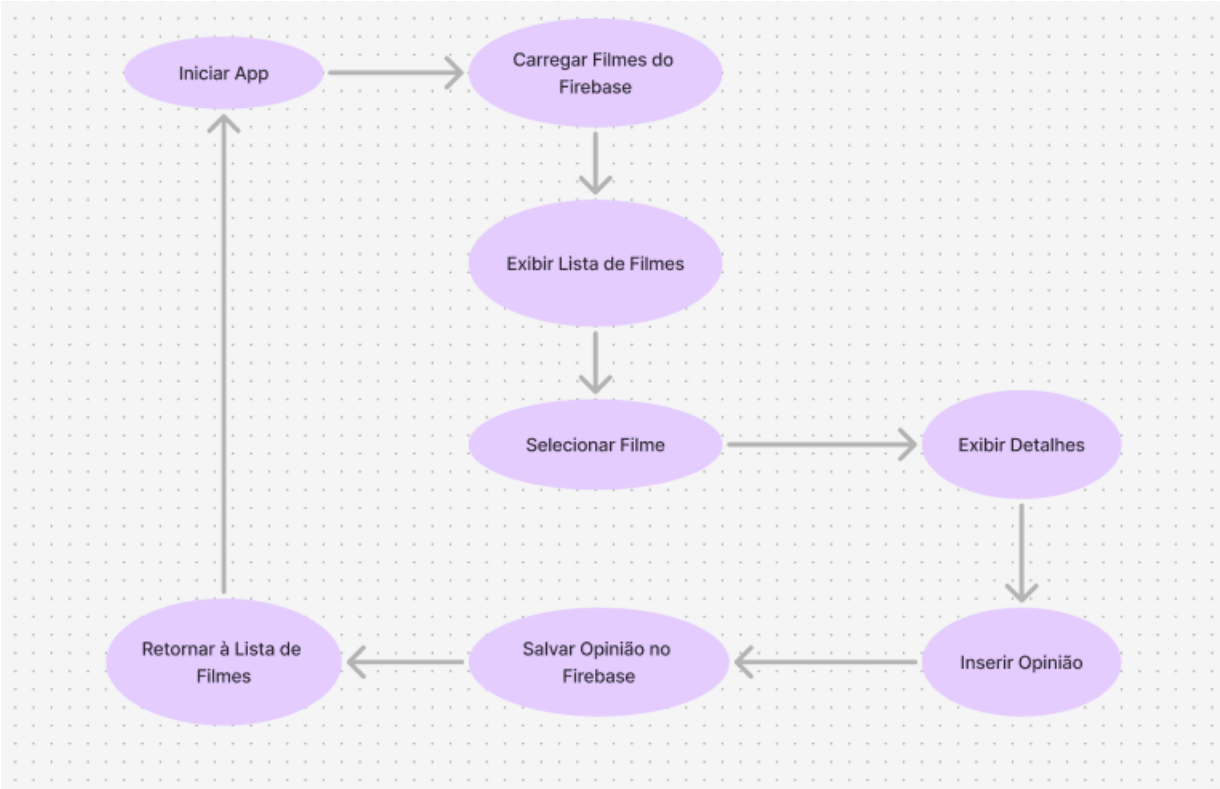
A metodologia proposta envolve a criação de uma série de aplicativos de teste, cada um utilizando um dos controladores de estado mencionados. Cada aplicativo será desenvolvido seguindo práticas padrão de desenvolvimento Flutter para garantir consistência. Após o desenvolvimento, será utilizado o Snapdragon Profiler para capturar e analisar as métricas de desempenho, como uso de CPU, memória e tempo de resposta. Este processo permitirá uma comparação objetiva entre os controladores de estado em termos de impacto na performance dos aplicativos *"Flutter"*.

3.1 Desenvolvimento dos Aplicativos

Para avaliar o desempenho dos controladores de estado no *"Flutter"*, os aplicativos foram implementados utilizando as abordagens *"BLoC"*, *"Provider"*, *"MobX"* e *"setState"*. A aplicação desenvolvida inclui uma lista de filmes e uma tela de detalhes, permitindo ao usuário salvar suas opiniões no Firebase Firestore. Cada um dos controladores foi empregado para gerenciar as atualizações de estado dessas funcionalidades, e o fluxograma apresentado na Figura 3 ilustra as

principais atividades e interações dentro da aplicação, detalhando o fluxo de dados e as transições de estado.

Figura 3 – Fluxo de Atividades da Aplicação



Fonte: Própria

3.1.1 Implementação com BLoC

Na abordagem utilizando o *"BLoC"*, a lógica que controla o funcionamento interno do aplicativo foi separada da parte visual que o usuário vê e interage. Para isso, foi adotado um método que utiliza "eventos" e "estados" como base para controlar as interações dos usuários e as mudanças nos dados exibidos. Isso significa que, por exemplo, cada ação do usuário, como carregar uma lista de filmes ou salvar uma opinião, era transformada em um evento que o *"BLoC"* processava. Com base nesses eventos, o estado do aplicativo era atualizado automaticamente. Essa abordagem trouxe uma estrutura organizada para o código, permitindo testar cada funcionalidade de forma independente, o que é muito útil em projetos mais complexos, onde é importante que cada parte do sistema funcione corretamente e de maneira isolada.

Apesar de suas vantagens, a implementação do *"BLoC"* também apresentou alguns desafios significativos. Foi necessário criar muitos eventos e estados separados para cada ação que o aplicativo realiza, o que acabou aumentando bastante a complexidade do código. Isso fez com que o processo de desenvolvimento se tornasse mais lento e mais complicado de aprender para novos desenvolvedores, principalmente no que diz respeito à configuração e sincronização correta

desses eventos. Além disso, a utilização de operações assíncronas, que são aquelas que acontecem em segundo plano sem travar o aplicativo, gerou uma curva de aprendizado maior. Foi preciso garantir que o fluxo de dados fosse controlado com precisão, de modo que a interface do usuário não fosse reconstruída de forma desnecessária, o que poderia afetar o desempenho do aplicativo. Portanto, embora a abordagem com *"BLoC"* ofereça um alto nível de organização, ela exige um entendimento profundo para evitar problemas de desempenho e garantir uma experiência de usuário fluida.

3.1.2 Implementação Provider

No desenvolvimento do aplicativo, foi utilizado um método chamado *"Provider"* junto com um recurso chamado *"ChangeNotifer"* para gerenciar como as informações eram compartilhadas e atualizadas dentro do sistema. Isso funcionou como uma forma de controle centralizado, permitindo que as mudanças feitas em certos dados fossem refletidas automaticamente nos elementos da tela que dependiam dessas informações. Dessa forma, quando o estado ou os dados eram alterados, a interface do aplicativo se atualizava de maneira automática, sem que fosse necessário recarregar tudo manualmente. Isso tornou o processo de desenvolvimento mais organizado e facilitou a manutenção do código, já que a lógica de atualização dos dados ficou concentrada em um único local.

No entanto, apesar dessa abordagem simplificar o gerenciamento de estado, surgiram desafios ao lidar com operações que precisavam acontecer simultaneamente. Por exemplo, quando o aplicativo buscava informações de um servidor ao mesmo tempo em que outras funções estavam sendo executadas, isso se tornava mais complicado, especialmente se vários componentes da interface dependiam do mesmo conjunto de dados. Nesses casos, poderia ocorrer um efeito de sobrecarga, onde todos os elementos da interface eram recarregados em vez de apenas aqueles que realmente mudaram. Para resolver essa questão, foram necessários ajustes específicos, garantindo que apenas os componentes que precisavam fossem atualizados. Isso foi ainda mais crítico em situações que exigiam dados em tempo real, como sensores ou mensagens de chat, onde era essencial que a atualização fosse imediata. Essas técnicas ajudaram a otimizar o desempenho, evitando o desperdício de recursos e garantindo uma melhor experiência para o usuário.

3.1.3 Implementação MobX

Na implementação do aplicativo, utilizou-se uma abordagem chamada *"MobX"*, que permite que certos dados sejam monitorados constantemente. Sempre que esses dados, conhecidos como observáveis, eram modificados, a interface era automaticamente atualizada, tornando essa técnica bastante útil para recursos que precisam de respostas em tempo real, como notificações ou atualizações instantâneas de informações.

Porém, surgiram desafios ao utilizar o *"MobX"*, especialmente devido à necessidade de gerar um código adicional usando uma ferramenta chamada *"build runner"*. Isso adicionou etapas extras ao desenvolvimento, o que tornou o processo mais lento e aumentou o risco de erros, principalmente se as configurações não estivessem corretas. Além disso, quando mudanças eram

feitas nos dados ou nas ações, era necessário regenerar o código, o que dificultava ajustes rápidos durante o desenvolvimento. Outro ponto crítico foi o elevado consumo de memória ao lidar com grandes volumes de dados, o que resultou em uma performance inferior comparada a outras técnicas de gerenciamento de estado, especialmente em listas maiores, onde o desempenho e a fluidez são essenciais para uma boa experiência do usuário.

3.1.4 Implementação SetState

A implementação com o *"setState"*, que é a ferramenta padrão do *"Flutter"* para gerenciar o estado dos componentes visuais, foi utilizada para controlar e atualizar a interface de forma direta e intuitiva. Basicamente, essa abordagem permite que partes específicas da tela sejam alteradas instantaneamente sempre que algo muda no aplicativo, como, por exemplo, exibir uma mensagem de carregamento ou atualizar um botão. O uso do setState mostrou-se bastante eficaz para situações onde é necessário controlar estados mais simples e localizados, ou seja, quando as mudanças afetam apenas uma parte pequena da tela e não a aplicação inteira. Essa simplicidade torna o desenvolvimento rápido, especialmente para funcionalidades básicas e temporárias, onde é preciso apenas ajustar detalhes visuais de maneira imediata.

No entanto, quando o aplicativo exigiu algo mais complexo, o uso do setState apresentou algumas limitações. Como essa ferramenta mistura a lógica do funcionamento do aplicativo com a forma como a interface é exibida, isso acabou deixando o código mais confuso e difícil de entender. No contexto do desenvolvimento de software, a ausência de separação entre as instruções que definem a lógica funcional de um aplicativo e aquelas que determinam sua interface visual pode acarretar desafios significativos para a manutenção e evolução do projeto. Essa integração excessiva em um único local de codificação tende a aumentar a complexidade, dificultando a identificação de problemas e a implementação de atualizações. Essa falta de separação entre o que o aplicativo faz e como ele se apresenta, especialmente em projetos que mudam com frequência, gerou um código que é mais propenso a erros. Isso também tornou mais difícil localizar e corrigir esses problemas, principalmente em funcionalidades mais elaboradas, onde há várias etapas envolvidas. Como resultado, essa abordagem exige um esforço maior tanto para manter o código organizado quanto para garantir que ele funcione corretamente em funcionalidades que necessitam de uma lógica de negócio mais complexa.

3.2 Análise Comparativa do Desempenho dos Controladores de Estado

Durante o desenvolvimento dos aplicativos utilizando o *framework Flutter*, foram implementadas diferentes abordagens para gerenciar o estado dos componentes visuais. Para isso, foram utilizados quatro controladores de estado distintos: *"BLoC"*, *"Provider"*, *"setState"* e *"MobX"*. Cada um desses métodos possui características únicas que impactam diretamente o desempenho da aplicação em termos de uso de memória, CPU, e tempo de resposta. Os testes realizados envolveram cenários práticos, como a troca de páginas, navegação entre listagens e o salvamento de dados, a fim de medir o impacto de cada abordagem no desempenho do sistema.

A escolha do controlador adequado é crucial para garantir que a aplicação seja responsiva e eficiente, especialmente em dispositivos móveis com recursos limitados.

3.2.1 Testes Realizados

Nos testes realizados, os dados foram normalizados utilizando a fórmula da média aritmética, com cinco repetições para cada cenário avaliado. Essa abordagem teve como objetivo obter uma análise mais equilibrada e confiável entre os diferentes controladores de estado testados. A normalização foi realizada somando todos os valores obtidos para cada métrica (uso de CPU, uso de memória e tempo de resposta) em cada repetição e dividindo pelo número total de repetições, garantindo que os resultados não fossem distorcidos por valores extremos ou picos ocasionais.

Os testes foram conduzidos em um dispositivo *Samsung Galaxy A22*, equipado com *Android* 13, 4 GB de memória RAM e um processador octa-core (2x 2.0 GHz Cortex-A75 + 6x 1.8 GHz Cortex-A55). Esse hardware foi escolhido por representar um perfil de dispositivo amplamente utilizado, permitindo avaliar o desempenho dos controladores em condições típicas de uso real. Com essa configuração e metodologia, foi possível identificar tendências consistentes e avaliar o desempenho médio de cada controlador de maneira robusta e realista.

A Tabela 4 apresenta os resultados obtidos para cada controlador, considerando as métricas de consumo de memória, uso de CPU e tempo de resposta. Os testes visaram analisar o desempenho dos diferentes controladores de estado em cenários práticos e frequentes no desenvolvimento de aplicações móveis, utilizando funcionalidades como troca de paginação, troca de página e salvamento de opiniões. Essas funcionalidades foram escolhidas por refletirem operações comuns em aplicativos que demandam atualizações dinâmicas da interface e interações constantes do usuário. Cada teste foi planejado de forma a medir o impacto de cada abordagem de gerenciamento de estado em métricas fundamentais, como consumo de memória, uso de CPU e tempo de resposta, permitindo uma análise comparativa detalhada entre os métodos avaliados.

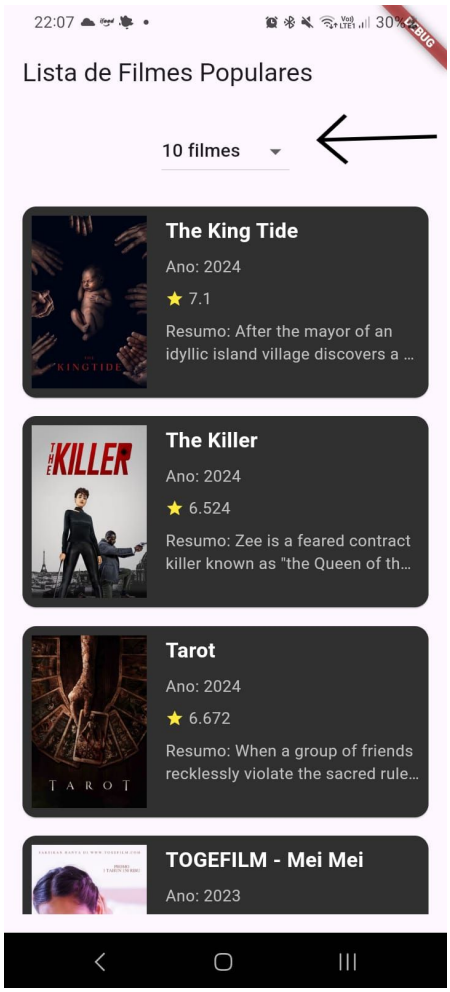
3.2.1.1 Troca de Paginação

Este teste consistiu em avaliar o desempenho da aplicação ao paginar listas extensas de dados nos intervalos de 10, 100 e 1000 itens, com foco na medição do tempo de resposta, do uso de memória e do consumo de CPU ao carregar novos conjuntos de itens na interface. A Figura 4 ilustra a interface utilizada no experimento, exibindo uma lista de filmes populares com a opção de selecionar diferentes quantidades de itens mostrados, como 10, 100 ou 1000. Os resultados revelaram diferenças significativas conforme apresentado na Tabela 1 entre os controladores de estado avaliados. O *"BLoC"* apresentou uso de memória entre 304,1 MB e 327 MB, consumo de CPU de 7% a 21% e tempo de resposta de 555 ms, destacando-se pela eficiência em listas volumosas. Já o *"Provider"* teve desempenho semelhante, com consumo de memória entre 301 MB e 330 MB, CPU de 3% a 23% e tempo de resposta de 543 ms, sendo uma abordagem equilibrada.

Por outro lado, o *"setState"* apresentou maior consumo de memória, variando de 325,2

MB a 360,5 MB, e um uso de CPU entre 10% e 27%, com tempo de resposta de 570 ms, o que aponta limitações em cenários mais exigentes. Já o *"MobX"* registrou o maior consumo de memória, entre 328,2 MB e 370 MB, e CPU de 12% a 28%, mas compensou com o menor tempo de resposta, atingindo 500 ms. Esse desempenho torna o *"MobX"* uma alternativa atrativa para aplicações que priorizam alta reatividade, ainda que com maior custo de recursos. Assim, os resultados evidenciam que a escolha do controlador deve ser guiada pelas demandas específicas da aplicação, equilibrando eficiência no uso de recursos e desempenho ideal para o usuário.

Figura 4 – Teste de Paginação



Fonte: Própria

Tabela 1 – Comparação de Desempenho do Teste Troca Paginação

Controlador	Memória (MB)	CPU (%)	Tempo de Resposta (ms)
setState	325,2 a 360,5	10 a 27	570
Provider	301,1 a 330	3 a 23	543
BLoC	304,1 a 327	7 a 21	520
MobX	328,2 a 370	12 a 28	500

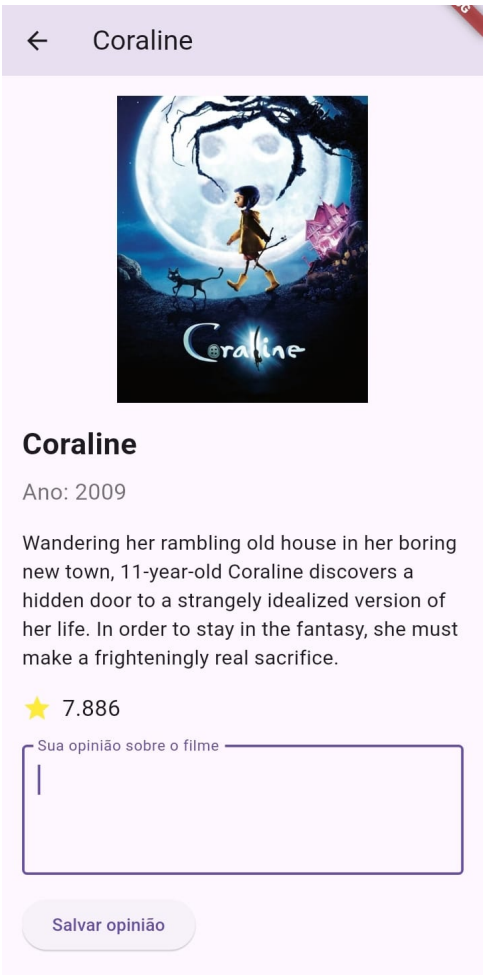
Fonte: Própria

3.2.1.2 Troca de Pagina

Neste teste isolado, focado na funcionalidade de troca de página no aplicativo, como ilustrado na Figura Figura 5, foram analisadas as transições entre diferentes telas, considerando o consumo de recursos, como memória e CPU, além da fluidez na navegação. Os resultados destacaram comportamentos distintos entre os controladores avaliados, conforme demonstrado na Tabela 2. O *"BLoC"* demonstrou o menor consumo médio de memória, variando entre 320 MB e 334,9 MB, e manteve o uso de CPU entre 5% e 15%, com um tempo de resposta de 245 ms, indicando boa eficiência geral. O *"Provider"* apresentou desempenho semelhante, com consumo de memória entre 325,7 MB e 338 MB, uso de CPU entre 6% e 17,3%, e um tempo de resposta de 281 ms.

O *"setState"*, por sua vez, exibiu maior consumo de memória, variando de 345,4 MB a 370,6 MB, além de um uso de CPU mais elevado, entre 7% e 18%, e um tempo de resposta de 283 ms, evidenciando limitações em cenários mais exigentes. O *"MobX"* destacou-se pelo menor tempo de resposta, de 208 ms, embora tenha registrado um consumo de memória entre 330,5 MB e 360,5 MB e um uso de CPU mais elevado, variando de 9% a 19%. Esses resultados isolados reforçam que o *"MobX"* é mais adequado para aplicações que priorizam alta reatividade, enquanto o *"BLoC"* e o *"Provider"* oferecem maior equilíbrio entre o consumo de recursos e o desempenho durante transições de tela.

Figura 5 – Pagina de Detalhes



Fonte: Própria

Tabela 2 – Comparação de Desempenho do Teste Troca de Pagina

Controlador	Memória (MB)	CPU (%)	Tempo de Resposta (ms)
setState	345,4 a 370,6	7 a 18	283
Provider	325,7 a 338	6 a 17,3	281
BLoC	320 a 334,9	5 a 15	245
MobX	330,5 a 360,5	9 a 19	208

Fonte: Própria

3.2.1.3 Salvar Opinião

O teste de salvamento de opinião foi realizado simulando o envio de dados para um banco remoto (Firebase Firestore), com a medição do tempo necessário para concluir a operação e atualizar a interface do usuário, como ilustrado na Figura Figura 6. Este procedimento teve como objetivo avaliar a eficiência dos controladores de estado na sincronização de dados locais com servidores externos, especialmente em operações que envolvem processamento assíncrono.

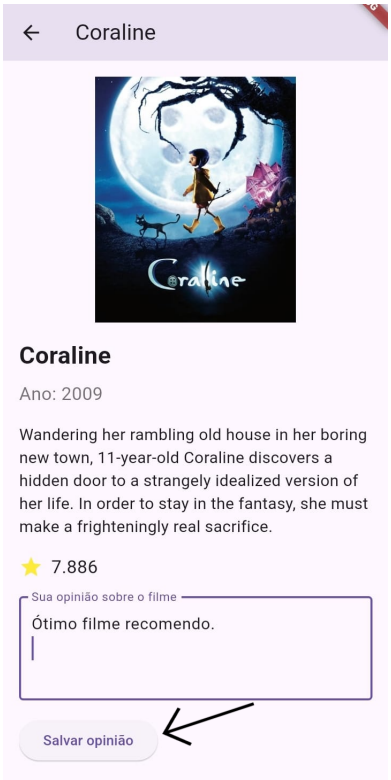
Os resultados isolados deste teste evidenciaram comportamentos distintos entre os controladores, conforme ilustrado na Tabela 3. O *"BLoC"* apresentou um consumo de memória que variou entre 330 MB e 362,4 MB, com uso de CPU entre 6,3% e 24%, e tempo de resposta médio de 615 ms. O *"Provider"*, por sua vez, teve um consumo de memória entre 334 MB e 357,7 MB, uso de CPU entre 7% e 23,5%, e tempo de resposta de 496 ms. O *"setState"* demonstrou um maior consumo de recursos, com memória variando entre 350 MB e 372,7 MB, uso de CPU de 10% a 25%, e tempo de resposta médio de 520 ms. Por fim, o *"MobX"* apresentou o menor tempo de resposta, com 336 ms, mas um consumo de memória que variou entre 340,5 MB e 380,5 MB, e uso de CPU entre 10% e 26%.

Tabela 3 – Comparação de Desempenho do Teste Salvar Opinião

Controlador	**Memória (MB)**	**CPU (%)**	**Tempo de Resposta (ms)**
setState	350 a 372,7	10 a 25	520
Provider	334 a 357,7	7 a 23,5	496
BLoC	330 a 362,4	6,3 a 24	615
MobX	340,5 a 380,5	10 a 26	336

Esses resultados demonstram que, embora o *"MobX"* tenha obtido o melhor desempenho em termos de tempo de resposta, seu consumo de recursos foi relativamente elevado. Por outro lado, o *"Provider"* destacou-se como a abordagem mais equilibrada, conciliando baixo consumo de memória e CPU com um tempo de resposta satisfatório.

Figura 6 – Teste Salvar Opinião



Fonte: Própria

3.2.2 Resultados Obtidos

Os testes mostraram que o *"setState"*, apesar de ser o método nativo e mais simples do *"Flutter"*, apresentou um uso elevado de CPU em cenários de grande volume de dados, como na navegação por listas extensas. Em um dos testes, ao paginar uma lista com 1000 itens, o uso de CPU chegou a picos de até 36%, com uma média em torno de 30%. Além disso, o tempo de resposta foi relativamente alto em cenários complexos, variando entre 600 ms e 680 ms. O consumo de memória também foi significativo, atingindo até 572,1 MB nos cenários mais intensos. Isso demonstra que o *"setState"* é mais adequado para funcionalidades simples e localizadas, onde o impacto no desempenho é minimizado.

Os resultados observados em testes com o controlador setState estão alinhados com os achados por Prayoga et al. (2021), que também indicou que esse método apresenta limitações significativas em cenários com grandes volumes de dados. Embora o setState seja eficiente em funções simples e localizadas, como visto nos testes realizados por Prayoga et al. (2021), seu desempenho se deteriora quando confrontado com interfaces complexas ou grandes quantidades de dados. Em seu estudo, Prayoga et al. (2021) apontou que, ao comparar o setState com outras abordagens, como o BLoC e o Provider, o uso da CPU e o tempo de resposta aumentam consideravelmente em cenários mais exigentes. Mesmo que o setState seja vantajoso em situações mais simples e em widgets de nível inferior, ele não se comporta de maneira eficiente em cenários de maior complexidade e maior volume de dados, como foi evidenciado na análise de listas extensas e navegação de dados pesados. Isso confirma que o setState é mais adequado para funcionalidades simples e de baixo custo computacional, sendo menos eficiente quando o sistema é submetido a demandas mais intensas de recursos.

Já o *"Provider"*, que utiliza um modelo de notificação eficiente para atualizar a interface do usuário, mostrou-se mais otimizado em termos de uso de memória do que o *"setState"*. Nos testes realizados, o *"Provider"* manteve um consumo médio de memória entre 269,8 MB e 551,3 MB, mesmo em operações complexas. O tempo de resposta foi relativamente menor, variando entre 420 ms e 680 ms, com uma média de aproximadamente 520 ms. No entanto, em operações que envolvem múltiplas atualizações simultâneas, como buscas em tempo real, o uso de CPU chegou a picos de 33%, o que pode causar lentidão em dispositivos mais limitados. Apesar disso, o *"Provider"* é uma escolha equilibrada para a maioria dos projetos, especialmente aqueles que requerem atualizações frequentes sem causar um grande impacto no desempenho.

Os resultados observados em testes com o controlador Provider estão alinhados com os achados por Prayoga et al. (2021), que também indicou que esse método apresenta vantagens significativas em termos de eficiência de memória e processamento. Embora o Provider tenha mostrado um desempenho otimizado em termos de uso de memória mesmo em operações complexas, seu tempo de resposta foi relativamente menor, variando entre 420 ms e 680 ms, com uma média de aproximadamente 520 ms, como observado nos testes realizados. Em seu estudo, Prayoga et al. (2021) apontou que o Provider se destaca em cenários que exigem atualizações frequentes sem um grande impacto no desempenho. No entanto, em situações de múltiplas atualizações simultâneas, como buscas em tempo real, o Provider apresentou picos de uso de CPU de até 33%, o que pode gerar lentidão em dispositivos mais limitados. Apesar disso, o

Provider continua sendo uma escolha equilibrada, especialmente para projetos que requerem a atualização constante da interface com eficiência, como foi evidenciado pelos resultados obtidos por Prayoga et al. (2021).

Por sua vez, o *"BLoC"* se destacou pela sua capacidade de lidar com operações mais complexas, especialmente aquelas que exigem uma clara separação entre a lógica de negócios e a interface visual. Após a normalização dos dados, o *"BLoC"* manteve um uso médio de CPU em torno de 20% e um uso de memória entre 283 MB e 515 MB, mesmo quando o número de itens da listagem chegou a 1000. O tempo de resposta foi consistentemente eficiente, variando entre 500 ms e 1111 ms, dependendo do cenário. Isso o torna ideal para aplicações que precisam escalar sem comprometer o desempenho. No entanto, a complexidade de implementação pode resultar em um tempo de desenvolvimento maior, o que pode ser um desafio para equipes menores ou projetos com prazos apertados.

O *"MobX"*, por outro lado, apresentou um desempenho misto em relação às outras abordagens. Foi bastante eficiente em atualizações reativas em tempo real, como na troca de páginas de listagem para detalhes, com um tempo de resposta que variou entre 210 ms e 310 ms. Contudo, em operações de navegação mais pesadas, o *"MobX"* teve um uso elevado de memória, atingindo picos de até 544,3 MB em listas com 1000 itens. O uso de CPU também foi um dos mais altos, com uma média de 32% e picos de até 39% em cenários de alta carga. Apesar disso, o *"MobX"* demonstrou tempos de resposta rápidos em situações que exigem alta reatividade, sendo uma escolha ideal para aplicações que demandam atualizações instantâneas, como notificações ou chats.

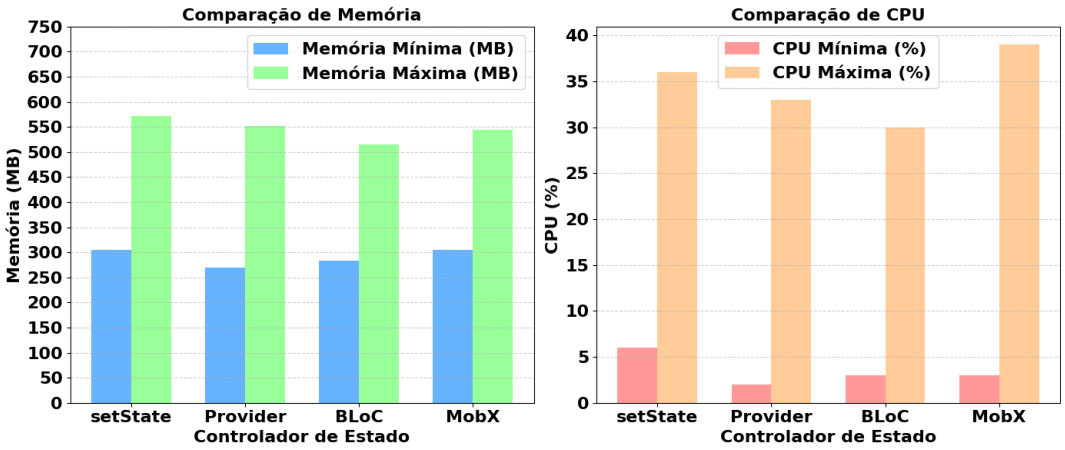
Tabela 4 – Comparação de Desempenho dos Controladores de Estado em Flutter

Controlador	**Memória (MB)**	**CPU (%)**	**Tempo de Resposta (ms)**
setState	304,6 a 572,1	6 a 36	600 a 680
Provider	269,8 a 551,3	2 a 33	420 a 680
BLoC	283,5 a 515,6	3 a 30	500 a 620
MobX	305,5 a 544,3	3 a 39	410 a 680

Fonte: Própria

Os testes realizados demonstraram que cada controlador de estado possui características que o tornam mais adequado para cenários específicos, dependendo das necessidades do projeto. O *"setState"* é mais indicado para aplicações simples, devido à sua implementação nativa e intuitiva, mas apresenta consumo elevado de memória e CPU em cenários complexos. Já o *"Provider"* se destacou como uma solução equilibrada para projetos que requerem atualizações frequentes, oferecendo um bom desempenho médio, embora possa enfrentar alguns desafios em operações intensivas, conforme ilustrado no gráfico da Figura 7, que compara o uso de memória e CPU entre os controladores.

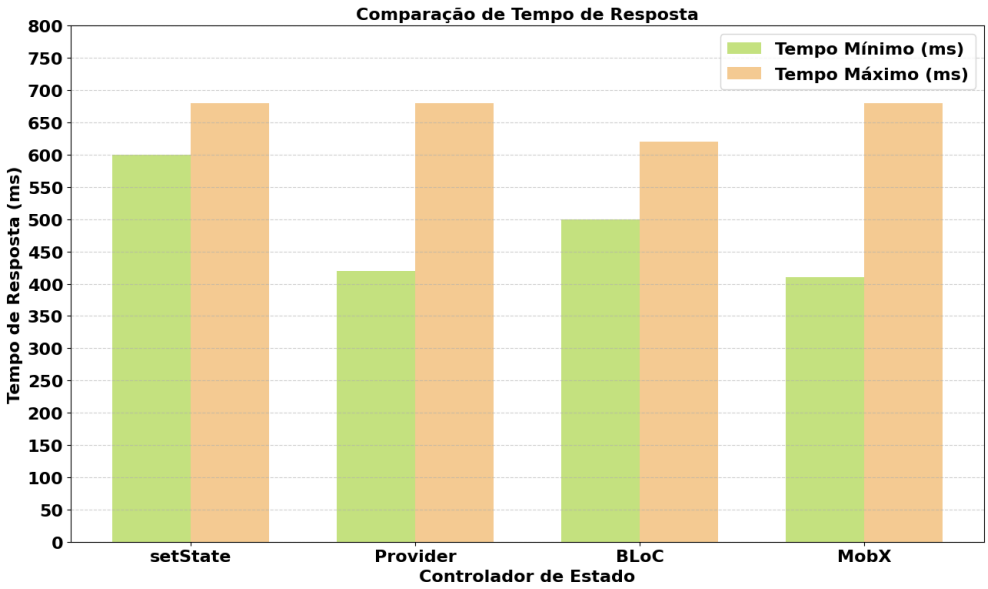
Figura 7 – Comparação do Uso de Memória e CPU entre Controladores de Estado



Fonte: Própria

Por outro lado, o *"BLoC"* mostrou-se ideal para aplicações que demandam escalabilidade e robustez, com bom controle do consumo de recursos, mesmo em cenários complexos. Finalmente, o *"MobX"* apresentou tempos de resposta significativamente rápidos como apresentado na Figura 8, sendo altamente indicado para aplicações que exigem alta reatividade, mas com o custo maior de memória e CPU em operações intensivas.

Figura 8 – Comparação do Tempo de Resposta entre Controladores de Estado



Fonte: Própria

4 Considerações finais

Este trabalho analisou e comparou o desempenho de quatro controladores de estado no *“Flutter”*: *“setState”*, *“Provider”*, *“BLoC”* e *“MobX”*. Através de uma metodologia prática e análise criteriosa, foi possível identificar as características de cada controlador em termos de consumo de memória, uso de CPU e tempo de resposta. Os resultados mostram que "setState"é mais eficaz para funcionalidades simples, enquanto "Provider"oferece um equilíbrio para aplicações com atualizações frequentes. "BLoC"se destacou pela robustez em cenários de alta complexidade, e "MobX"apresentou excelente reatividade em tempo real, embora com maior consumo de recursos em operações intensivas.

Os desafios enfrentados durante o estudo, como a complexidade do *“BLoC”* e o consumo elevado do *“MobX”*, ressaltam a importância de adaptar os controladores às demandas específicas de cada projeto. Apesar disso, o trabalho conseguiu atingir os objetivos propostos, fornecendo uma base sólida para que desenvolvedores escolham a abordagem mais eficiente conforme o tipo de aplicação. A análise também contribuiu para a compreensão das vantagens e limitações de cada método, incentivando a adoção de práticas que otimizem desempenho e experiência do usuário.

Como continuidade deste estudo, sugere-se investigar novos controladores de estado ou explorar soluções híbridas que combinem as vantagens das abordagens existentes. Também seria relevante analisar o impacto desses controladores em dispositivos com hardware mais limitado, proporcionando insights para aplicações em contextos de maior restrição. Espera-se que as conclusões deste trabalho sirvam de referência prática e teórica, promovendo avanços no desenvolvimento de aplicações *“Flutter”*.

Performance Analysis of State Controllers in the Flutter Framework

Igor Dal Cero Rocha* Jorge Luis Boeira Bavaresco†

2024

Abstract

This work presents a comparative analysis of the performance of different state controllers in the Flutter framework: *"setState"*, *"Provider"*, *"BLoC"*, and *"MobX"*. State management is a crucial element in mobile application development, especially in scenarios requiring frequent and responsive user interface updates. The research was motivated by the question: "Which state controller offers the best performance for a Flutter application?" and aimed to evaluate the impact of each approach on memory consumption, CPU usage, and response time. To achieve this, test applications were developed that implemented navigation and state update functionalities using each controller. Performance was measured using the Snapdragon Profiler, analyzing metrics such as memory consumption, CPU usage, and execution time. The results revealed that *"setState"* is more suitable for simple functionalities, while *"Provider"* proved efficient in applications requiring frequent updates. *"BLoC"*, in turn, stood out in high-complexity scenarios due to its ability to separate business logic from the interface. On the other hand, *"MobX"* demonstrated excellent performance in applications demanding high reactivity, despite its higher resource consumption in intensive scenarios. The comparative analysis contributes to understanding the advantages and limitations of each controller, helping developers choose the most appropriate approach for different types of applications. This study provides a valuable theoretical and practical basis, aiming to optimize performance and user experience in projects developed with Flutter.

Key-words: Flutter. State management. Performance analysis.

*Dados do autor.
†Orientador do trabalho (dados).

Referências

ALMEIDA, J. V. R. de. O que é gerenciamento de estados no flutter e principais ferramentas. *Alura Artigos*, 2024. Acessado em 27 de junho de 2024. Disponível em: <https://www.alura.com.br/artigos/gerenciamento-de-estados-flutter-principais-ferramentas>. Citado na página 5.

ANGELOV., F. *Documentação BLoC Business Logic Component*. 2023. Disponível em: <https://pub.dev/packages/bloc>. Acesso em 07 Abril 2024. Citado na página 3.

DART, G. *Documentção Dart*. 2024. Disponível em: <https://https://dart.dev/overview>. Acesso em 07 Abril 2024. Citado na página 3.

Flutter. *Simple app state management*. 2023. Disponível em: <https://docs.flutter.dev/data-and-backend/state-mgmt/simple>. Acessado em: 13 de abril de 2024. Citado na página 4.

LTD., H. T. P. *State Management in Flutter: Provider, Riverpod, and BLoC*. 2023. Disponível em: <https://hupp.tech/blog/programming/state-management-in-flutter-provider-riverpod-and-bloc/>. Acesso em 07 Abril 2024. Citado na página 4.

MOZ, A. D. *Gerenciamento de Estado MobX*. 2020. Disponível em: <https://medium.com/android-dev-moz/gerenciamento-de-estado-mobx-f7934c96682e>. Acesso em 2 de junho de 2024. Citado na página 5.

PRAYOGA, R. R. et al. Performance analysis of bloc and provider state management library on flutter. *Jurnal Mantik*, v. 5, n. 3, p. 1591–1597, 2021. Citado 3 vezes nas páginas 6, 16 e 17.

QUALCOMM. *Snapdragon Profiler*. 2024. Disponível em: <https://docs.qualcomm.com/bundle/publicresource/topics/80-71528-1/snapdragon_profiler.html?product=1601111740010426>. Acessado em: 21 de novembro de 2024. Citado na página 7.

RABELO, R. Gerenciamento de estado no flutter e o pacote provider. *Medium*, agosto 2020. Acesso em 07 Abril 2024. Citado na página 4.

SANTOS, J. V. P. *Flutter com MobX. Um estudo sobre o uso de MobX*. 2020. Disponível em: <https://medium.com/flutter-comunidade-br/flutter-com-mobx-c0f4762fbd1a>. Acesso em 2 de junho de 2024. Citado na página 5.

TASHILDAR, A. et al. Application development using flutter. *International Research Journal of Modernization in Engineering Technology and Science*, v. 2, n. 8, p. 1262–1266, 2020. Citado na página 2.

ZULISTIYAN, M.; ADRIAN, M.; WIBOWO, Y. F. A. Performance analysis of bloc and getx state management library on flutter. *Journal of Information System Research (JOSH)*, v. 5, n. 2, p. 583–591, 2024. Citado 3 vezes nas páginas 3, 4 e 6.