Sate Management in Flutter

1st Fabio Sinabell
Department for Smart And Interconnected Living
Software Architecture and Patterns
Hagenberg, Austria
s2410455007@students.fh-hagenberg.at

2nd Thomas Wagner
Department for Smart And Interconnected Living
Software Architecture and Patterns
Hagenberg, Austria
s2410455017@students.fh-hagenberg.at

3rd Lukas Zandomeneghi
Department for Smart And Interconnected Living
Software Architecture and Patterns
Hagenberg, Austria
s2410455009@students.fh-hagenberg.at

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert.**

I. Introduction

Flutter has emerged as a leading framework for cross-platform application development [1], [3]. It offers a declarative UI paradigm and a unified codebase for multiple platforms [4], [5]. While this approach accelerates development and promotes visual consistency [1], it also intensifies the challenge of managing application state as projects grow [6]. In Flutter, user interfaces rebuild in response to state changes [5]. The lack of a prescribed architectural pattern has led to many state-management solutions [6]. At small scales, state can often be handled locally within widgets with minimal architectural overhead. However, at larger scales, applications must coordinate asynchronous data flows, domain logic, and UI updates across multiple features and teams [2], [3]. This makes state management a central concern for maintainability, testability, and long-term evolution [2].

The Flutter ecosystem reflects this challenge through the coexistence of numerous state management approaches, including Provider, Riverpod, Redux, MobX, GetX, Cubit, and the BLoC (Business Logic Component) pattern [6], [7]. Each approach has different assumptions about separation of concerns, reactivity, and developer ergonomics [5]. Consequently, architectural decisions are often influenced by community conventions, tutorials, or anecdotal experience rather than systematic evidence [2], [3]. This situation is particularly problematic for applications developed over extended lifecycles because early architectural choices can constrain scalability, hinder onboarding, and increase the cost of change [2], [3].

Of these approaches, BLoC has received a lot of attention because of its explicit modeling of events and states, as well as its focus on unidirectional data flow and separating business logic from presentation concerns [5]. Advocates argue that these characteristics make BLoC particularly suitable for large, long-lived Flutter applications [5]. Critics, however, point to its perceived verbosity and steep learning curve [5]. Despite its widespread adoption, academic and technical literature on BLoC is fragmented, and its advantages and limitations are rarely evaluated in a structured, comparative manner against alternative state management solutions [7].

This paper argues that a rigorous, literature-based comparison of Flutter state management approaches is necessary to inform architectural decision-making at scale [1]–[3]. The paper posits that the strengths and weaknesses of BLoC can only be meaningfully understood in relation to competing approaches and evaluation criteria that extend beyond developer preference to include learnability, testability, performance, scalability, and maintainability [5], [7]. To support this claim, the paper synthesizes peer-reviewed publications, white papers, and high-quality technical reports on BLoC and alternative state management patterns within the Flutter ecosystem.

This work contributes a structured comparison of BLoC with other widely used Flutter state-management approaches. This comparison is grounded in existing literature rather than isolated case studies [6], [7]. A running Flutter application serves as a didactic artifact that illustrates core concepts, such as events, states, and repositories, and contextualizes maintainability-related observations reported in the literature [5]. However, it does not serve as an empirical evaluation in its own right. Furthermore, the paper distills the findings into practical guidance on when adopting BLoC is advantageous and identifies areas where future empirical research is needed to substantiate or challenge existing assumptions [5]. Thus, the paper aims to support researchers and practitioners in making informed, evidence-based decisions about state management in Flutter applications [1]–[3].

II. Background

In Flutter, state management is closely tied to the framework's declarative UI model [8], [9]. In this model, the visual representation of an application depends on its current state. Whenever the state changes, the framework triggers the rebuild of widgets, propagating the updates through the widget tree [10], [11]. While this design simplifies reasoning about UI rendering, it shifts the complexity toward how state is represented, shared, and mutated over time [5], [12]. As Flutter applications evolve to include more than small, self-contained screens, state increasingly spans multiple widgets, features, and asynchronous data sources, such as network services or local persistence layers [11]. The lack of a standard architectural pattern has led to a variety of state management approaches, each with different trade-offs regarding abstraction, reactivity, and separation of concerns [12].

Within this landscape, the BLoC (Business Logic Component) pattern is an approach that strictly separates presentation from business logic [15]. In BLoC, the UI does not directly mutate the state. Instead, it emits events representing user interactions or external triggers. The BLoC processes these events and contains the application's business logic, reacting by producing new states [15]. These states are then exposed as streams to which the UI subscribes to rebuild itself accordingly [15]. This unidirectional flow—from events to states—provides a clear, explicit model of how data moves through the system and how changes are propagated [15]. By relying on streams, BLoC naturally integrates with Dart's asynchronous programming model and supports complex workflows involving debouncing, event transformation, and event composition [15], [16].

Cubit is a simplified variant of the BLoC pattern [17]. Although Cubit preserves the idea of separating business logic from the UI, it does not explicitly model events. Rather, state changes are triggered by method calls on the Cubit itself, with each method emitting a new state [17]. This design simplifies the implementation for straightforward state transitions [17]. Consequently, Cubit is often considered a pragmatic alternative to the full BLoC architecture when additional event handling structure is unnecessary [17].

Other state-management approaches in Flutter adopt different design philosophies. The Provider approach builds on Flutter's widget mechanism to enable dependency injection and state propagation through the widget tree [5], [13]. It emphasizes simplicity and close integration with Flutter's core concepts, making it suitable for small- to medium-sized applications [5]. However, it leaves architectural discipline largely to the developer. Riverpod builds on the ideas of Provider, decoupling state from the widget tree [18].

Redux introduces a global, immutable state container, which is inspired by the Flux architecture [14]. In this architecture, all state changes are expressed as actions that are processed by pure reducers. While this approach offers strong predictability and time-travel debugging, it requires additional boilerplate code [14]. MobX, by contrast, follows a reactive programming paradigm based on observable state and automatic dependency tracking [14]. MobX aims to minimize boilerplate code and allows state changes to propagate implicitly. GetX combines state management, routing, and dependency injection in a single framework, prioritizing convenience and minimal syntax [7].

Taken together, these approaches demonstrate that there is no single dominant solution for Flutter state management, but rather a spectrum of patterns ranging from lightweight, implicit reactivity to highly structured, explicit data flows [5], [12]. BLoC and Cubit occupy a position within this spectrum that emphasizes clarity, unidirectional data flow, and testability [5]. Understanding these differences provides the conceptual foundation for the comparative analysis undertaken in the remainder of this paper.

III. Comparison Prototype

To enable a concrete and comparable discussion of Flutter state management paradigms, a prototypical application was implemented using multiple approaches. The prototype models a minimal but representative authentication workflow, including login, logout, asynchronous repository access, and navigation between screens. This scope was chosen because it combines user interaction, asynchronous state changes, and cross-screen state propagation, which are common requirements in real-world Flutter applications.

Each variant of the prototype implements the same user interface, navigation flow, and repository abstraction. The authentication repository simulates an external data source and is shared across all implementations to avoid confounding architectural differences. Only the state management layer differs between variants. This constraint ensures that observed differences arise primarily from the paradigms themselves rather than from unrelated design decisions. The full implementation is publicly available as an open-source repository[1].

The prototype includes implementations using BLoC, Cubit, Provider, Riverpod, Redux, MobX, and GetX. For each approach, the application exposes identical user-facing behavior: validation of input, a loading state during authentication, a success state containing user data, and a logout operation that resets application state. Navigation logic is intentionally kept minimal to focus on how each paradigm models and propagates state changes.

Rather than serving as a benchmark for performance or developer productivity, the prototype functions as a didactic artifact. It enables a systematic comparison of architectural structure, explicitness of data flow, and maintainability-related properties discussed in the literature. In particular, it provides a concrete basis for examining claims about verbosity, scalability, and separation of concerns, with BLoC serving as the primary reference architecture for comparison.

IV. Comparative Analysis of State Management Paradigms

This section outlines the core principles of each state management paradigm represented in the prototype, followed by a comparative discussion of their strengths and limitations as observed in the implementation. The analysis emphasizes BLoC and relates alternative approaches to it.

[1]https://github.com/LukasZando/saap_bloc_project

A. Core Paradigms

BLoC (Business Logic Component)[2] models application behavior through explicit events and immutable states. State transitions are handled in response to events, enforcing unidirectional data flow and a strict separation between presentation and business logic. All side effects are isolated from the UI layer.

Cubit[3] is a streamlined variant of BLoC that removes the explicit event layer. State changes are triggered directly through method calls, reducing boilerplate while retaining controlled, reactive state emission.

Provider[4] exposes mutable state objects to the widget tree using inherited widgets, commonly relying on `ChangeNotifier` for update propagation. It integrates closely with Flutter's widget lifecycle and favors simplicity over structural rigidity.

Riverpod[5] generalizes the Provider model by decoupling state access from the widget hierarchy. Providers are immutable descriptions of state dependencies, enabling improved testability and clearer dependency management.

Redux[6] centralizes application state into a single immutable store. State changes occur exclusively through dispatched actions processed by pure reducer functions, enforcing a highly rigid unidirectional data flow.

MobX[7] follows a reactive programming model in which observable state automatically triggers updates in dependent observers. State mutations are permitted through annotated actions, with reactivity handled implicitly. MobX further relies on code generation to transform annotated classes into reactive stores. This mechanism simplifies the definition of observable state but introduces an additional compilation step and dependency on build tooling, which may influence development workflow and project configuration.

GetX[8] combines state management, dependency injection, and navigation into a unified framework. It emphasizes minimal boilerplate and direct state mutation with reactive updates. Furthermore, the fact that it is a whole framework, not just a state management library, can be seen as a strength, but also means overkill for simple applications, if only state management is needed.

B. Prototype-Based Comparison

Within the prototype, BLoC exhibits the highest degree of structural explicitness. All state transitions are centralized and traceable, making the flow of data and side effects immediately visible. This clarity comes at the cost of verbosity, particularly for simple interactions such as form input validation. However, the explicit modeling aligns well with the prototype's asynchronous authentication logic and facilitates reasoning about edge cases.

Cubit demonstrates that much of BLoC's structural benefit can be retained with reduced syntactic overhead. In the prototype, Cubit provides a more concise implementation while still maintaining clear ownership of state transitions. This makes it suitable for medium-sized features where full event modeling is unnecessary.

Provider and MobX yield the most concise implementations. For the prototype's limited scope, both approaches are easy to implement and intuitive to follow. However, state mutations can occur from multiple locations, which reduces transparency. As the prototype grows beyond a single feature, reasoning about state dependencies becomes more difficult compared to BLoC-based approaches.

Riverpod improves upon Provider by making dependencies explicit and removing reliance on widget context for state access. In the prototype, this results in cleaner separation of concerns and improved testability. While still less rigid than BLoC, Riverpod offers a balanced trade-off between structure and flexibility.

Redux enforces a level of rigor comparable to BLoC but at a global scale. In the prototype, this rigidity appears excessive relative to the application's size. The requirement to define actions and reducers for all state changes increases cognitive overhead without providing proportional benefits for localized state.

GetX enables the fastest implementation of the prototype due to its minimal boilerplate and integrated navigation. However, the implicit nature of state updates and reliance on framework conventions reduce architectural transparency. This makes it difficult to reason about state flow as the application scales.

C. Guidance for Practical Use

The comparison suggests that the suitability of a state management paradigm depends strongly on application scale, architectural requirements, and team conventions. BLoC is particularly appropriate for large, long-lived Flutter applications in which explicit state transitions, unidirectional data flow, and strict separation of concerns are required. Its verbosity introduces overhead for small features but provides long-term benefits in maintainability, testability, and onboarding as system complexity grows.

Cubit and Riverpod offer viable alternatives when a similar degree of structure is desired with reduced boilerplate. Cubit is well suited for feature-scoped state with limited interaction complexity, while Riverpod provides flexible dependency management and improved testability without enforcing a fully event-driven model. Both approaches can serve as intermediate solutions between lightweight state handling and the rigor of BLoC.

Provider and MobX prioritize simplicity and developer ergonomics. They are effective in scenarios where state logic remains localized and relatively stable. However, as demonstrated in the prototype, implicit state mutations and distributed

[2]https://bloclibrary.dev/
[3]https://bloclibrary.dev/bloc-concepts/#cubit
[4]https://docs.flutter.dev/data-and-backend/state-mgmt/simple
[5]https://riverpod.dev/
[6]https://pub.dev/documentation/flutter_redux/
[7]https://mobx.netlify.app/
[8]https://pub.dev/packages/get

update logic can make it harder to reason about application behavior as features accumulate, which may limit scalability in larger codebases.

Redux enforces a highly centralized and rigid architecture that can be beneficial for applications requiring strict global state consistency and traceability. In the context of the prototype, this level of rigor introduces substantial boilerplate relative to the problem size and is most justifiable in applications with complex cross-feature coordination.

GetX differs from the other approaches in that it provides not only state management but also navigation and dependency injection as part of a unified framework. This integration can reduce implementation effort but also increases coupling to framework-specific conventions. As a result, GetX represents a broader architectural commitment rather than a lightweight state management choice, which should be carefully evaluated in projects with long-term maintainability or framework independence requirements.

Acknowledgment

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression “one of us (R. B. G.) thanks ...”. Instead, try “R. B. G. thanks...”. Put sponsor acknowledgments in the unnumbered footnote on the first page.

References

[1] G. Jošt and V. Taneski, ”State-of-the-Art Cross-Platform Mobile Application Development Frameworks: A Comparative Study of Market and Developer Trends,” Informatics, vol. 12, no. 2, p. 45, 2025. DOI: 10.3390/informatics12020045

[2] D. Zou and M. Y. Darus, ”A Comparative Analysis of Cross-Platform Mobile Development Frameworks,” in 2024 International Symposium on Computer Science and Intelligent Controls (ISCI), 2024. DOI: 10.1109/isci62787.2024.10667693

[3] A. Souha, L. Benaddi, C. Ouaddi, and A. Jakimi, ”Comparative analysis of mobile application Frameworks: A developer’s guide for choosing the right tool,” Procedia Computer Science, vol. 239, pp. 712-720, 2024. DOI: 10.1016/j.procs.2024.05.071

[4] M. Kumar, ”Evaluating Modern Android Frameworks: A Comparative Study of Flutter, Kotlin Multiplatform, Jetpack Compose, and React Native,” Indian Scientific Journal Of Research In Engineering And Management, vol. 09, no. 01, 2025. DOI: 10.55041/ijsrem48732

[5] Sanghmitra, ”The State Management Dilemma: BLoC vs. Provider in Modern Flutter Development,” International Journal of Scientific Research in Computer Science, Engineering and Information Technology, vol. 10, no. 5, pp. 270-279, 2024. DOI: 10.32628/cseit241051027

[6] M. Zulistiyan, M. Adrian, and Y. F. A. Wibowo, ”Performance Analysis of BLoC and GetX State Management Library on Flutter,” Journal of Information System Research (JOSH), vol. 5, no. 2, pp. 683-692, 2024. DOI: 10.47065/josh.v5i2.4698

[7] A. A. D. Jatnika, M. A. Akbar, and A. Pinandito, ”Comparative Analysis of the Use of State Management in E-commerce Marketplace Applications Using the Flutter Framework,” JITeCS (Journal of Information Technology and Computer Science), vol. 8, no. 2, pp. 184-195, 2023. DOI: 10.25126/jitecs.202382557

[8] D. Slepnev, ”State management approaches in Flutter,” Haaga-Helia University of Applied Sciences, 2021. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/355086/Dmitrii_Slepnev.pdf

[9] E. Windmill, *Flutter in Action*. New York, NY, USA: Simon & Schuster, 2020.

[10] R. R. Prayoga, A. Syalsabila, G. Munawar, and R. Jumiyani, ”Performance Analysis of BLoC and Provider State Management Library on Flutter,” Oct. 2021. [Online]. Available: https://scispace.com/papers/performance-analysis-of-bloc-and-provider-state-management-4mrtvrzlir

[11] F. Cheng, ”State Management,” in *Build Native Cross-Platform Apps with Flutter*, Apress, 2019, pp. 227–246, doi: 10.1007/978-1-4842-4982-6_10.

[12] M. Szczepanik and M. Kedziora, ”State Management and Software Architecture Approaches in Cross-platform Flutter Applications,” in Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, SCITEPRESS, 2020. [Online]. Available: https://www.scitepress.org/Papers/2020/94116/94116.pdf

[13] T. Tran, ”Flutter: Native Performance and Expressive UI/UX,” Project Report, San Jose State University, 2020.

[14] L. Ventura, ”Analysis of Redux, MobX and BLoC and how they solve the state management problem,” M.S. thesis, Politecnico di Milano, 2022. [Online]. Available: https://www.politesi.polimi.it/handle/10589/190202

[15] Bloc Library, ”Architecture,” 2026. [Online]. Available: https://bloclibrary.dev/architecture/. [Accessed: Jan. 13, 2026].

[16] Google, ”Asynchronous Programming: Streams,” Dart, 2026. [Online]. Available: https://dart.dev/tutorials/language/streams. [Accessed: Jan. 13, 2026].

[17] Bloc Library, ”Cubit,” 2026. [Online]. Available: https://bloclibrary.dev/bloc-concepts/#cubit. [Accessed: Jan. 13, 2026].

[18] Riverpod, ”Riverpod Documentation,” 2026. [Online]. Available: https://riverpod.dev/. [Accessed: Jan. 13, 2026].