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 [12], [14]. It offers a declarative UI paradigm and a unified codebase for multiple platforms [15], [16]. While this approach accelerates development and promotes visual consistency [12], it also intensifies the challenge of managing application state as projects grow [17]. In Flutter, user interfaces rebuild in response to state changes [16]. The lack of a prescribed architectural pattern has led to many state-management solutions [17]. 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 [13], [14]. This makes state management a central concern for maintainability, testability, and long-term evolution [13].

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 [17], [18]. Each approach has different assumptions about separation of concerns, reactivity, and developer ergonomics [16]. Consequently, architectural decisions are often influenced by community conventions, tutorials, or anecdotal experience rather than systematic evidence [13], [14]. 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 [13], [14].

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 [16]. Advocates argue that these characteristics make BLoC particularly suitable for large, long-lived Flutter applications [16]. Critics, however, point to its perceived verbosity and steep learning curve [16]. 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 [18].

This paper argues that a rigorous, literature-based comparison of Flutter state management approaches is necessary to inform architectural decision-making at scale [12]–[14]. 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 [16], [18]. 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 [17], [18]. 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 [16]. 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 [16]. Thus, the paper aims to support researchers and practitioners in making informed, evidence-based decisions about state management in Flutter applications [12]–[14].

II. Background

In Flutter, state management is closely tied to the framework's declarative UI model [19], [20]. 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 [21], [22]. While this design simplifies reasoning about UI rendering, it shifts the complexity toward how state is represented, shared, and mutated over time [16], [23]. 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 [22]. 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 [23].

Within this landscape, the BLoC (Business Logic Component) pattern is an approach that strictly separates presentation from business logic [26]. 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 [26]. These states are then exposed as streams to which the UI subscribes to rebuild itself accordingly [26]. This unidirectional flow—from events to states—provides a clear, explicit model of how data moves through the system and how changes are propagated [26]. By relying on streams, BLoC naturally integrates with Dart's asynchronous programming model and supports complex workflows involving debouncing, event transformation, and event composition [26], [27].

Cubit is a simplified variant of the BLoC pattern [28]. 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 [28]. This design simplifies the implementation for straightforward state transitions [28]. Consequently, Cubit is often considered a pragmatic alternative to the full BLoC architecture when additional event handling structure is unnecessary [28].

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 [16], [24]. It emphasizes simplicity and close integration with Flutter's core concepts, making it suitable for small- to medium-sized applications [16]. However, it leaves architectural discipline largely to the developer. Riverpod builds on the ideas of Provider, decoupling state from the widget tree [29].

Redux introduces a global, immutable state container, which is inspired by the Flux architecture [25]. 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 [25]. MobX, by contrast, follows a reactive programming paradigm based on observable state and automatic dependency tracking [25]. 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 [18].

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 [16], [23]. BLoC and Cubit occupy a position within this spectrum that emphasizes clarity, unidirectional data flow, and testability [16]. Understanding these differences provides the conceptual foundation for the comparative analysis undertaken in the remainder of this paper.

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

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," 2013, arXiv:1312.6114. [Online]. Available: https://arxiv.org/abs/1312.6114

[9] S. Liu, “Wi-Fi Energy Detection Testbed (12MTC),” 2023, gitHub repository. [Online]. Available: https://github.com/liustone99/Wi-Fi-Energy-Detection-Testbed-12MTC

[10] “Treatment episode data set: discharges (TEDS-D): concatenated, 2006 to 2009.” U.S. Department of Health and Human Services, Substance Abuse and Mental Health Services Administration, Office of Applied Studies, August, 2013, DOI:10.3886/ICPSR30122.v2

[11] K. Eves and J. Valasek, “Adaptive control for singularly perturbed systems examples,” Code Ocean, Aug. 2023. [Online]. Available: https://codeocean.com/capsule/4989235/tree

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

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

[21] 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

[22] 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.

[23] 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

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

[25] 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

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

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

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

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