Evaluating State Management Frameworks in Flutter: A Comparative Analysis of Bloc, Riverpod, and GetX

Professional development for engineers and English (TDA518) — 25-03-2024

1st Casper Christiansson
Chalmers University of Technology
Gothenburg, Sweden

2nd Harry Denell
Chalmers University of Technology
Gothenburg, Sweden

3rd Rikard Roos
Chalmers University of Technology
Gothenburg, Sweden

***Abstract*—This paper conducts a comparative analysis of three prominent state management frameworks within the Flutter ecosystem: Bloc, Riverpod, and GetX. It evaluates these frameworks based on ease of state definition, manipulation, and consumption, highlighting their advantages and disadvantages. This is done through the lens of what constitutes a good framework. Through this analysis, the paper aims to provide valuable insights for both novice and experienced Flutter developers in selecting the most suitable state management framework, thereby enhancing their development process and application performance.**

I. Introduction

Flutter, a Google-developed open-source UI toolkit, enables the creation of visually appealing, high-performance apps for mobile, web, and desktop from one codebase [1]. It leverages the Dart programming language, aimed at client development, to support rapid iteration with features like null safety and hot reload [2]. Flutter operates on a widget-centric framework, where widgets serve as the fundamental components of the user interface [3]. Widgets are categorized into two types: stateful and stateless. Stateful widgets are dynamic and can undergo changes during runtime without necessitating a component reinitialization, whereas stateless widgets cannot [3]. Since Flutter's user interface is composed of a widget tree, setting a state of a parent widget, would cause all its children to have to rerender, which will result in performance drawbacks. Without a proper state managing architecture, this process can become unmaintainable [4]. One way to solve this is to use a state management solution.

This paper aims to explore and analyze the efficacy of various state management solutions within the Flutter ecosystem. The subjects of analysis are Business Logic Component (Bloc) [5], Riverpod [6], and GetX [7], being some of the most popular state management solutions [8]. Through this analysis the paper aims to guide Flutter developers, both beginners and experts, in selecting the most optimal state management solution within the Flutter and Dart ecosystem.

II. Effective State Management Frameworks

State management is a software engineering concept that deals with how the application's state changes with various events [9], and is crucial for creating dynamic applications. Effective state management ensures that the application behaves consistently with user interaction, enhances performance as well as improves code quality and testability [10]. Figure 1 below presents the general state management architecture.

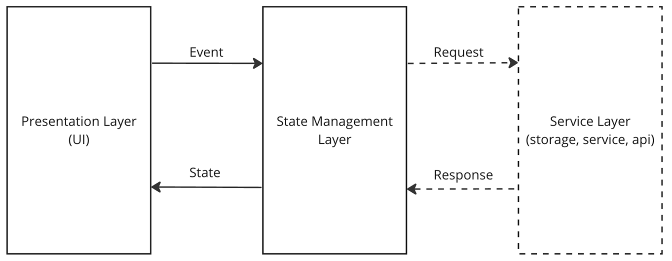

Figure 1. Diagram of the state management process, showing data flow between the UI, state management, and optional service layers.

The presentation layer triggers events, based on user interactions, that inform the state management layer of required actions to perform on the service layer, such as data retrieval, database updates, or calculations. The state management layer processes these requests and updates the application state with the results or completion status.

A framework, in the context of software engineering, can be described as a foundation that developers can use and build upon to create software more efficiently [11]. It comprises a set of components and patterns that offer a reusable design for a system [11]. A good framework should be easy to use and the developer should not need to know how it is implemented [11]. This abstraction allows for a higher level of productivity by simplifying the development process and enabling a focus on the application's functionality rather than its implementation. This applies to any framework, including state management frameworks.

In software development, there exists a nuanced distinction between the concepts of frameworks and libraries [12]. Despite this, the terms are sometimes used interchangeably [12]. In discussing Flutter state management solutions, there's a noticeable inconsistency in the usage of the terms frameworks and libraries. Riverpod and GetX are often referred to as frameworks, whereas Bloc is typically referred to as a library. In this paper, Bloc, Riverpod and GetX will all be categorized as state management frameworks, following Johnson's [11] broad framework definition.

A. State Management Frameworks in Flutter

Bloc emphasizes simplicity, speed, and testability [5]. It allows for the development of complex applications through the composition of smaller, reusable components [5]. Bloc aims to make state changes predictable by enforcing a structured and consistent method for managing state transitions [5].

Riverpod enhances the principles of Provider [13] with a more flexible and scalable approach, enabling developers to safely access and modify the application's state from anywhere in the codebase [6]. It introduces a compile-time safe way of managing state, ensuring errors are caught before runtime [6].

GetX emphasizes light-weight and simplicty [7]. It allows developers to manage state, navigate between screens, and manage dependencies with minimal boilerplate code [7].

III. Framework Analysis Criteria

To conduct a focused analysis, we defined specific criteria to look at, tying into the efficiency and abstraction principles in a good framework [11].

State Definition: We examined how state is defined and encapsulated within each framework. Less verbose generally implies that developers can define stateful components faster and easier, adhering to the principle of ease of use.

State Manipulation: We compared how to update the state, including the declaration of events, actions, or methods that trigger state changes. A good framework should allow for clear and concise state manipulation. How the state is changed should be abstracted away, and the developer should be given clear instructions on how the framework handles state manipulation.

State Consumption: We analyzed how each framework allows widgets to consume state changes. This includes the mechanisms to listen to state changes and rebuild the UI accordingly. The goal was to identify which framework minimizes the code necessary for this behavior, while encapsulating this behaviour into best practices to follow.

IV. Framework Analysis

The analysis was applied on a codebase that manages user authentication, one for each framework, covering the login and logout processes, along with error handling for potential issues. The codebase is available on GitHub. [1]

[1]Link to the GitHub repository: https://github.com/harryden/TDA518ConferencePaper/tree/main/conference_paper/lib

A. Bloc

State Definition: In the Bloc framework, states and events work together to manage an app's flow. States represent the program's conditions, like initializing, loading, or handling errors, while events are triggers, like user actions, that prompt state changes within the Bloc. This setup organizes the program's reactions to inputs or changes, in a clear and structured way, making the overall state management intuitive and controlled. The state definition process is streamlined and clearly defined, not needing to know how its implemented. On the other hand the state definition of Bloc is verbose, therefore requiring more time to implement.

State Manipulation: State changes are triggered by events, which in this instance include AuthenticationLogin for login attempts and AuthenticationLogout for logging out. The AuthBloc class is designed to listen for these events, initiating interactions with the AuthService through asynchronous functions to manage authentication processes. Upon the initiation of an event, the Bloc transitions through states in a predictable sequence. Bloc's structured approach to state manipulation, where state changes are triggered by events and managed through a predictable sequence, exemplifies the principle of abstraction. Developers do not need to understand the entire flow but can focus on defining clear events and state transitions.

State Consumption: State consumption is facilitated through the use of specialized widgets and the injection of Blocs into the widget tree. Widgets such as BlocBuilder, BlocListener, and BlocConsumer can be used, reacting to state changes by either triggering UI rebuilds (BlocBuilder) or executing specific actions (BlocListener), with BlocConsumer offering a combination of both functionalities. The use of these widgets for state consumption allows focus on the application's functionality rather than its implementation. These widgets abstract away the complexities of state changes and UI updates, making the framework easier to use.

B. Riverpod

State Definition: States are defined by first defining an abstract class, for example ‘AuthenticationState‘. Different states such as loading, error, logged in, and logged out, are then created by extending the abstract class. They might contain additional parameters such as a string or a User object, for example.

State Manipulation: State manipulation is achieved by interacting with a StateNotifier associated with a Provider. In the authentication example, the ‘logout‘ and ‘login‘ methods are called on the ‘AuthStateNotifier‘ from the ‘AuthenticationScreen‘ widget. The ‘ref.read‘ method is used to obtain a reference to the ‘AuthStateNotifier‘, specifically its notifier component, which manages the state logic for authentication. By invoking the ‘logout‘ or ‘login‘ methods, the application triggers a state change process within the ‘AuthStateNotifier‘.

State Consumption: State consumption involves reacting to changes in the state managed by a StateNotifier associated with a Provider. In the authentication example, a ‘StateNotifierProvider‘ is subscribed to using the ‘ref.watch‘ method. When the state changes, the UI will be rebuilt depending on what the state is. Riverpod has no division between different widgets to use depending on if the new state should result in a behaviour or a UI update. A lack of abstraction, might infer and hinder the development process of writing UI code that can handle different, potentially complex states.

C. GetX

State Definition: State is defined through observable variables, denoted by .obs. The use of .obs for defining state variables is less verbose compared to some other state management solutions, which may require more boilerplate code to achieve reactivity. This characteristic aligns with the efficiency principle by allowing developers to define state with minimal code, also enhancing readability and maintainability. On the other hand, complex behaviour through states, might be harder to model with GetX, due to state changes only being reflected in updated variables, where for some states the developer might not want the variable to have a value at all, therefore having to set the variable to null.

State Manipulation: State manipulation is achieved with minimal code. The AuthController class manages the authentication state, including isLoggedIn, isLoading, and error, with observable variables (.obs). State changes are triggered via methods like login and logout. These methods directly update the observable variables, which in turn, trigger the UI to update. The amount of code required to declare and trigger these state changes is minimal, adhering to the principle of clear and concise state manipulation, without need to know the concrete implementations. By encapsulating the internal state changes, only allowing the user to make state changes through defined methods, the principle of the developer not needing to know how the state change is implemented, is adhered.

State Consumption: State consumption is facilitated through the Obx widget, which automatically rebuilds its child widgets in response to observable changes. This mechanism is showcased in the AuthenticationScreen class, where UI components are conditionally rendered based on the authentication state. However, in scenarios where the requirement is not to return a widget but to execute a function, relying on workaround methods like ‘addPostFrameCallback‘ could deviate from the principle of abstraction. GetX should ideally provide a more integrated solution for this.

V. Discussion

Bloc’s architecture emphasizes robustness and scalability by using events and states to manage the app’s flow in a structured way, which facilitates modularity and maintainability at the expense of more boilerplate code. GetX, in contrast, minimizes boilerplate code with its use of observable variables for state definition and manipulation. However, while its simplicity offers rapid development, it may struggle in handling complex state behavior with the same precision as Bloc.

Riverpod shares many similarities with Bloc across the three analysed criteria. The major difference is how it handles accessibility. In Riverpod, instances of the state management layer (providers) are globally accessible throughout the entire widget tree. This accessibility simplifies state sharing and manipulation, but also introduces risks associated with global state management, such as unintended modifications that can lead to bugs. However, in Bloc, instances of the state management layer (Blocs), are accessible only within the widget subtree where they are provided. While this restriction in Bloc enhances encapsulation and reduces the risk of unintended state modifications, it can make sharing state across separated parts of the app more challenging, which could lead to more complex code.

In terms of state consumption, Bloc offers a more refined approach by providing widgets for listening to state changes, rebuilding the UI in response to these changes, or performing both actions concurrently. This allows UI components to react to state changes seamlessly without needing to resort to workaround methods, as seen in GetX and Riverpod. This streamlines the development process, making the codebase more readable and maintainable.

In summary, Bloc restricts state management to widget subtrees for encapsulation, while Riverpod offers global accessibility. GetX prioritizes rapid development with less boilerplate but may falter in handling complex states as precisely as Bloc, which provides dedicated widgets for seamless UI state consumption and management. Although each framework offers its unique strengths, for Flutter developers seeking a state management framework, Bloc likely provides an ideal mix of robustness, abstraction, and verbosity. GetX, while efficient, may fall short in managing complex states, and Riverpod’s absence of predefined widgets and its approach to global state management might introduce challenges.

References

[1] Google Inc., ”Flutter - Build apps for any screen,” 2024. [Online]. Available: https://flutter.dev/ (accessed on: 2024-02-14).

[2] Google Inc., ”Dart programming language — Dart,” 2024. [Online]. Available: https://dart.dev/ (accessed on: 2024-02-14).

[3] Google Inc., ”Building user interfaces with Flutter,” 2024. [Online]. Available: https://docs.flutter.dev/ui (accessed on: 2024-02-20).

[4] S. Boukhary and E. Colmenares, ”A clean approach to flutter development through the flutter clean architecture package,” *2019 International Conference on Compuational Science and Compuational Intelligence (CSCI)*, Las Vegas, NV, USA, 2019, pp. 1115-1120. [Online]. Available: https://doi.org/10.1109/CSCI49370.2019.00211, Accessed on: 2024-02-14.

[5] F. Angelov., ”Bloc library,” 2024. [Online]. Available: https://bloclibrary.dev/#/ (accessed on: 2024-02-14).

[6] R. Rousselet., ”Riverpod: A simple way to manage state in flutter,” 2024. [Online]. Available: https://riverpod.dev/ (accessed on: 2024-02-14).

[7] Flutter Community., ”get: A package for flutter state management,” 2024. [Online]. Available: https://pub.dev/packages/get (accessed on: 2024-02-14).

[8] Google Inc. ”pub.dev,” 2024. [Online]. Available: https://pub.dev/packages?q=state+management&sort=like (accessed on: 2024-03-19).

[9] B. Wu, ”Analyze and Compare the State Management Patterns of VUE,” in *2023 IEEE International Conference on Image Processing and Computer Applications (ICIPCA)*, Changchun, China 2023, pp. 1279-1282. [Online]. Available: https://doi.org/10.1109/ICIPCA59209.2023.10257967, Accessed on: 2024-03-18.

[10] E. Elrom, ”State Management” in *React and Libraries: Your Complete Guide to the React Ecosystem*, E. Elrom. Berkely, CA, USA: Apress, 2021, ch. 5, pp. 115-147. [Online]. Available: https://doi.org/10.1007/978-1-4842-6696-0, Accessed on: 2024-02-15.

[11] R. E. Johnson, ”Frameworks = (components + patterns),” *Commun. ACM*, vol. 40, no. 10, pp. 39-42, Oct. 1997, doi: 10.1145/262793.262799.

[12] GeeksForGeeks. ”Software Frameworks vs Library,” 2023. [Online]. Available: https://www.geeksforgeeks.org/software-framework-vs-library/ (accessed on: 2024-03-19).

[13] Flutter Community., ”Provider: A package for Flutter state management,” 2023. [Online]. Available: https://pub.dev/packages/provider (accessed on: 2024-02-15).