DOI: https://doi.org/10.30837/IST-2024.P1.119

Exploring the methods for using state machines to optimize and enhance cross-platform mobile applications

Vladyslav Martynov[1,*,†], Ihor Shubin[1,†] and Victoria Skovorodnikova[1,†]

[a] *Kharkiv National University of Radio Electronics, Nauky ave., 14, Kharkiv, 61166, Ukraine*

Abstract

Nowadays, contemporary mobile apps require robust state management, particularly in cross-platform frameworks such as React Native. Finite State Machines (FSMs) provide a structured methodology for managing state transitions, enhancing stability, performance, and testability in complex applications. This article examines the advantages of FSMs and their practical applications in cross-platform development, with a focus on React Native.

Keywords

FSM, React, React-Native, XState

1. Introduction

As modern mobile applications become increasingly embedded in users' daily routines, delivering responsive and efficient functionality is paramount. These applications serve a vast array of purposes—quick access to information, seamless communication, task management, and beyond—placing high demands on developers to continually enhance functionality, performance, and reliability. Cross-platform frameworks like React-Native empower developers to deploy applications across multiple platforms, streamlining development and reducing resource expenditure. Yet, cross-platform solutions introduce distinct challenges, particularly in state management.

Effective state management remains a central challenge in cross-platform development. Complex applications, especially those with multifaceted user interactions, asynchronous tasks, and extensive use of device resources, rely on smooth state transitions to ensure consistent behavior. Without a robust architecture for state management, applications risk crashes, unpredictable behaviors, and a diminished user experience.

Finite State Machines (FSMs) offer a systematic approach to addressing these challenges. By rigorously defining all possible states and permissible transitions, FSMs enforce predictable and stable application behavior, proving especially useful in managing navigation flows, handling asynchronous operations, and structuring complex logic. Within mobile applications, FSMs excel at optimizing scenarios driven by external and internal events, such as user authentication and server requests, where precise state control is essential.

A major advantage of FSMs is their ability to impose a well-defined structure on application control logic, minimizing errors and simplifying state tracking. This structure enhances testability, allowing developers to independently assess each state and transition and

Information Systems and Technologies (IST-2024), November 26-28, 2024, Kharkiv, Ukraine

* Corresponding author.

† These authors contributed equally.

vladyslav.martynov@nure.ua (V. Martynov); ihor.shubin@nure.ua (I. Shubin); viktoriia.skovorodnikova@nure.ua (V. Skovorodnikova)

0009-0003-5665-2323 (V. Martynov); 0000-0002-1073-023X (I. Shubin); 0000-0003-0436-4197 (V. Skovorodnikova)

proactively identify potential issues. Furthermore, FSMs contribute to performance optimization by streamlining transitions and reducing unnecessary computations, a critical consideration for cross-platform solutions where execution speed directly impacts user satisfaction.

2. Review of the literature

The concept of state is very familiar. Consider a toaster: initially, the toaster is in the off state; when you push the lever down, a transition is made to the on state and the heating elements are turned on; finally, when the timer expires, a transition is made back to the off state and the heating elements are turned off.

A finite state machine (FSM) consists of a set of states s_j and a set of transitions between pairs of states s_i, s_j . A transition is labeled condition/action: a condition that causes the transition to be taken and an action that is performed when the transition is taken. FSMs can be displayed as a state diagram Fig.1.

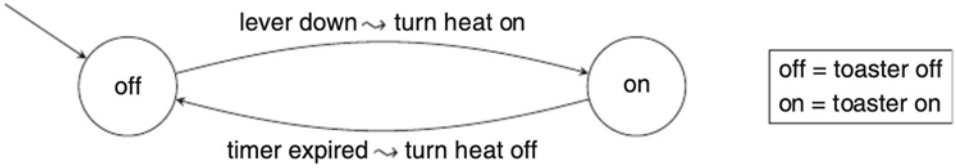

Figure 1: State diagram

A state is denoted by a circle labeled with the name of the state. States are given short names to save space and their full names are given in a box next to the state diagram. The incoming arrow denotes the initial state. A transition is shown as an arrow from the source state to the target state. The arrow is labeled with the condition and the action of the transition. The action is not continuing; for example, the action turn left means set the motors so that the robot turns to the left, but the transition to the next state is taken without waiting for the robot to reach a specific position [1].

React Native is an open source framework for building Android and iOS applications using React and the app platform's native capabilities. With React Native, you use JavaScript to access your platform's APIs as well as to describe the appearance and behavior of your UI using React components: bundles of reusable, nestable code [2].

Unlike hybrid frameworks that run within a web view, React Native compiles to native components, providing applications with a look and feel similar to those built with platform-specific languages like Swift for iOS or Kotlin for Android.

However, managing complex user flows in React Native can be challenging due to platform-specific differences, which impact component behavior, navigation patterns, and resource management. These differences often require additional logic to handle varied platform responses, such as asynchronous tasks, permission handling, and native animations, which can lead to increased complexity and potential inconsistencies in user experience. Addressing these challenges effectively requires a structured approach to state management, particularly when handling intricate navigation and interactive flows across platforms.

Finite State Machines (FSMs) offer an effective solution for such state management challenges by enforcing a strict and predictable sequence of states and transitions. In scenarios involving complex workflows—like multi-step forms, authentication flows, or dynamic content rendering—FSMs provide developers with a clear framework for defining each state and its permissible transitions. This structure mitigates the unpredictability that can arise from asynchronous tasks or varying platform behavior, helping to ensure consistent responses to user actions across iOS and Android.

By using FSMs within React Native applications, developers gain the ability to encapsulate complex user interactions in a way that is both testable and maintainable. Each state transition

is explicitly defined, which reduces the likelihood of unintended behaviors and facilitates debugging. In cross-platform environments where maintaining uniformity is essential, FSMs provide a reliable approach for handling intricate application states, enhancing both the predictability and resilience of mobile applications.

3. Experiments

In this experiment, the basic authentication screen will be implemented with advanced, specialized library called Xstate and then will be compared with default React Native implementation.

XState is a state management and orchestration solution for Javascript and Typescript apps. It uses event-driven programming, state machines, statecharts, and the actor model to handle complex logic in predictable, robust, and visual ways. XState provides a powerful and flexible way to manage application and workflow state by allowing developers to model logic as actors and state machines. It integrates well with React, Vue, Svelte, and other frameworks and can be used in frontend, backend, or wherever Javascript runs [3].

XState enhances the clarity and robustness of state management through its advanced syntax, which allows developers to encapsulate and manage state logic directly within the machine configuration. XState's internal context functions as a dedicated data store, managing state-specific information such as validation flags, input values, and error messages. This encapsulation minimizes reliance on external state management solutions, consolidating all relevant logic and data within a single, cohesive unit.

4. Results

To evaluate the performance of both the React Native default implementation and XState implementations, we used three key metrics: the React Native UI thread, the JS thread, and FPS drop. The UI and JS thread metrics measure the responsiveness of the application by tracking frame rates on each thread, with higher values indicating smoother performance. FPS drop, on the other hand, reflects any decrease in frames per second during interactions, with lower values indicating fewer disruptions in user experience. These metrics provide a comprehensive view of each implementation's ability to handle complex interactions smoothly.

Based on Table 1 metrics, XState demonstrates superior performance across multiple dimensions. In the XState implementation, both the UI and JS threads maintain a steady average of 60 frames per second (FPS). This consistent performance indicates that XState efficiently handles state transitions and user interactions without overloading either thread. The minimal FPS drop (2-3) further suggests that XState is capable of processing complex interaction patterns smoothly, keeping the application responsive and ensuring a seamless user experience even under heavy usage conditions. This level of stability is crucial in maintaining an interactive flow, especially on an authentication screen where real-time validation and feedback are essential.

In contrast, the React Native default implementation shows performance limitations, with the UI thread averaging 55 FPS and the JS thread at 50 FPS. This reduction in frame rates suggests that it introduces more load on the system, potentially due to less optimized transition handling or higher overhead in managing state changes. The FPS drop is also significantly higher, ranging from 10 to 12 FPS during intensive interactions. This larger drop indicates that default implementation experiences performance bottlenecks under complex user actions, resulting in occasional stutters and a less fluid experience. Such delays can negatively impact user perception, especially in scenarios where users expect immediate feedback, such as entering login credentials.

In summary, XState's structured approach and optimized handling of state transitions provide a clear advantage in terms of performance, maintaining high frame rates on both UI and JS threads and minimizing FPS drops.

Table 1
Performance metrics

Metrics	XState	React Native default implementation
UI Thread	60	55
JS Thread	60	50
FPS Drop	2-3	10-12

5. Conclusion

This study demonstrates the effectiveness of using finite state machines to manage complex interactions within mobile applications, particularly in cross-platform frameworks like React Native. By implementing an authentication screen in two variants — React Native default implementation and the XState library—we explored the performance, maintainability, and user experience benefits of each approach.

The XState implementation proved to be superior in terms of performance, with consistently high frame rates and minimal FPS drops, ensuring a smooth and responsive experience. Its advanced abstractions, such as context handling, hierarchical state management, and visual state charts, make it especially well-suited for applications with intricate interaction flows. The ability to visualize state charts not only aids in understanding and debugging complex workflows but also facilitates collaboration among team members and provides a form of living documentation as the application evolves.

React Native default implementation, demonstrated some limitations in handling resource-intensive transitions and maintaining consistent frame rates under heavy user interactions. Although it offers basic state management, the overhead associated with managing states manually makes it less suitable for highly interactive or complex forms, especially when compared to the modular and declarative approach offered by XState.

In conclusion, XState is a powerful tool for managing complex forms and dynamic interactions across various applications. Its structured approach to state management makes it versatile enough for a wide range of use cases, including multi-step forms, onboarding flows, authentication processes, and any scenario requiring predictable state transitions. The library's modular design allows developers to maintain a clean codebase, improve performance, and enhance user experience. As mobile applications continue to grow in complexity, adopting a well-defined state management solution like XState can play a critical role in building robust, scalable, and maintainable applications.

Future work could explore extending this approach to more intricate applications involving real-time updates, multi-user collaboration, or machine learning-driven interactions. Furthermore, the potential for combining XState with other tools in the React Native ecosystem, such as gesture handlers and animations, opens up possibilities for crafting highly interactive and intuitive user experiences.

References

[1] Finite State Machine URL: http://surl.li/xbonql (date of access 20.11.2024)
[2] React-Native URL: http://surl.li/bwzllc (date of access 18.11.2024)
[3] XState URL: https://stately.ai/docs/xstate (date of access 15.11.2024)