

STATE MACHINES IN WEB DEVELOPMENT: MANAGING UI COMPLEXITY

Modern web applications are characterized by increasingly complex user interfaces (UIs) that function not merely as presentation layers but as dynamic behavioral systems. Contemporary UIs respond to user actions, asynchronous network requests, validation procedures, retries, permission checks, animations, and real-time updates [2]. In domains such as banking, healthcare, logistics, and education, interface errors may lead to financial losses, incorrect decisions, or erosion of user trust [1]. Under such conditions, the key challenge in front-end development is not just rendering data correctly but also ensuring that the interface behaves predictably and consistently in every possible scenario.

The purpose of this work is to substantiate the applicability of finite state machines (FSMs) as a formal model for managing UI complexity in modern web applications and to demonstrate how this approach enhances predictability, reduces logical errors, and improves maintainability.

A finite state machine represents system behavior as a finite set of explicitly defined states and controlled transitions between them [4]. At any given moment, the system is in exactly one state, and transitions to other states occur only through predefined events. Applied to user interfaces, this model enables developers to describe not only data conditions but the lifecycle of interaction itself. For example, a data-loading component can be formally represented by states such as "idle", "loading", "success", and "error". The transition from "idle" to "loading" is triggered by a user action; from "loading" to "success" or "error" by the corresponding server response; and from "error" back to "loading" by a retry action. Any transition not explicitly defined – such as moving directly from "idle" to "success" without a loading phase – is excluded by design.

The main problem this approach addresses is the limitations of traditional UI state handling based on multiple boolean flags or loosely connected state variables. Three typical categories of issues arise in such implementations.

First, impossible states may occur when contradictory conditions are simultaneously true. For instance, a component may indicate both loading and success, or display data together with an error message, especially under race conditions or unstable network behavior. Such inconsistencies are difficult to detect and often manifest only in edge cases. A state machine eliminates this risk by enforcing state mutual exclusivity: the interface can occupy only one well-defined state at a time.

Second, transition logic in conventional implementations is frequently scattered across event handlers, asynchronous callbacks, lifecycle hooks, and store actions. This fragmentation complicates reasoning about a screen's overall behavior and makes maintenance more error-prone [3]. In contrast, a state machine centralizes all transitions within a single declarative structure. The full behavioral map becomes transparent not only to developers but also to testers and designers, facilitating collaborative work and systematic verification.

Third, modern web interfaces operate in asynchronous environments where events may arrive in unpredictable order. Examples include delayed API responses, repeated user interactions, or updates from external sources. Without explicit control, such events may trigger unintended transitions or overwrite valid states. FSMs address this challenge by specifying which events are valid in each state and defining how invalid or unexpected events are handled (ignored, blocked, or redirected) [4]. This structured handling significantly reduces the risk of race conditions and inconsistent UI behavior.

It is important to distinguish state machines from general state management tools such as Redux, Pinia, or similar frameworks. Conventional state management focuses on storing and updating data, whereas a state machine governs the sequence and legality of transitions between

behavioral states. In practical applications, these approaches are complementary: the data store stores application data, while the state machine orchestrates the interaction logic.

A representative example is a media uploader with multiple stages: selecting files, adding metadata, uploading, completion, cancellation, and error handling. Without an explicit behavioral model, such a feature quickly becomes a collection of loosely connected conditions. By modeling it as a state machine with clearly defined states (e.g., SelectingFiles, AddingMetadata, Uploading, Uploaded, Canceled, Error) and allowed transitions, the interface gains structural clarity. The UI always "knows" its current phase and which actions are permissible, thereby simplifying both implementation and testing.

Despite their advantages, state machines are not universally required. For simple components with minimal branching logic, ordinary state variables are sufficient. FSMs introduce an additional abstraction layer and require developers to think in terms of states, events, and transitions. Therefore, their use is most justified in complex, multi-step, asynchronous, or error-prone processes where correctness and predictability are critical.

The practical value of the proposed approach lies in improving reliability, reducing the number of hidden logical defects, and enhancing code readability and maintainability. By making implicit behavioral assumptions explicit, state machines transform UI development from managing scattered conditions to designing controlled interaction processes. This shift supports higher software quality standards and strengthens collaboration within development teams.

In conclusion, finite-state machines provide a structured, disciplined framework for managing UI complexity in modern web development. By explicitly defining valid states and transitions, they prevent impossible states, centralize behavioral logic, and ensure predictable handling of asynchronous events. Their application is particularly effective in complex user flows where reliability and clarity are essential.

References

1. Alaoui R. L., Nfaoui E. H. Deep learning for vulnerability and attack detection on web applications: a systematic literature review. *Future internet*. 2022. Vol. 14, no. 4. P. 118. URL: <https://doi.org/10.3390/fi14040118>.
2. An MLLM-assisted web crawler approach for web application fuzzing / W. Yang et al. *Applied sciences*. 2025. Vol. 15, no. 2. P. 962. URL: <https://doi.org/10.3390/app15020962>.
3. Semantic similarity-based mobile application isomorphic graphical user interface identification / J. Cheng et al. *Mathematics*. 2023. Vol. 11, no. 3. P. 527. URL: <https://doi.org/10.3390/math11030527>.
4. Formalizing UML state machines for automated verification – A survey / É. André et al. *ACM computing surveys*. 2023. URL: <https://doi.org/10.1145/3579821>.