

State Management Models in Modern Frontend Applications

Vladislav Meshcheryakov

Senior Front-End Developer, Coexwalf OU, Tallinn, Estonia
ORCID ID: 0009-0007-5316-9351;

ABSTRACT: *The article examines state management models in modern frontend applications as an independent architectural layer of client-side logic. The evolution of state management approaches is analyzed: from local component state and state transfer through the component hierarchy to context-based, centralized, modular, server-state, form-state, and hybrid models. Particular attention is paid to the structuring of client-side data and the separation of UI state, domain entities, server state, forms, navigation parameters, and derived values. The mechanisms of state consistency in client-side and asynchronous user scenarios are considered, including the distribution of responsibility between local, domain, and server state, caching, invalidation, optimistic updates, rollback, and handling of request race conditions. It is shown that the choice of a state management model affects maintainability, testability, interface predictability, and the resilience of a frontend system as its functional complexity increases.*

KEYWORDS: *Asynchronous Scenarios, Client Architecture, Frontend Application, Server State, State Management, UI State.*

I. INTRODUCTION

Modern frontend applications are functionally complex, rely on intensive interaction with server-side APIs, and include many asynchronous operations that require a consistent user interface under changing data conditions. Unlike earlier web interfaces, modern client-side applications handle user actions, form validation, routing, caching, loading states, error processing, and synchronization with remote sources. Under these conditions, state management becomes a key architectural mechanism that determines application predictability, user scenario stability, and further software development.

The relevance of studying state management models is determined by the increasing complexity of component interactions, data transfer, and side effects as a frontend system grows. Poorly organized client-side state reduces code readability,

complicates testing, weakens team collaboration, and increases maintenance costs. Therefore, the choice of a state management model should be treated not only as the use of a library or framework, but as an architectural decision related to responsibility separation, business logic structuring, asynchronous process management, and frontend scalability. The aim of this article is to analyze modern state management approaches and determine their impact on the maintainability, reliability, and engineering stability of client-side systems.

II. METHODS

The study was conducted as a review-analytical work using systemic, comparative, and architectural-logical analysis. Its empirical basis included recent scientific publications and industry reports on frontend development. Systemic analysis

was used to examine state management as an independent layer of frontend architecture connected with component structure, client-side logic, server interaction, and user scenarios. The comparative method was applied to assess local, centralized, modular, and hybrid models in terms of maintainability, testability, predictability, and team development. Architectural-logical analysis helped identify the main categories of client-side state, including UI state, domain data, server state, form state, navigation parameters, and derived values. Frontend engineering practices were also generalized to systematize approaches to asynchronous synchronization, caching, invalidation, optimistic updates, rollback, error handling, and state recovery.

III. EVOLUTION OF STATE MANAGEMENT APPROACHES IN FRONTEND DEVELOPMENT

The increasing functional load of the frontend layer has changed the requirements for state management: it is no longer limited to storing temporary UI values, but has become an architectural mechanism for working with local, domain, and server data, API interaction, and the consistency of user scenarios [1]. In general terms, the development of state management approaches can be represented as a gradual transition from server-oriented interfaces and local component state to centralized, modular, and differentiated models of frontend architecture (**Fig. 1**).

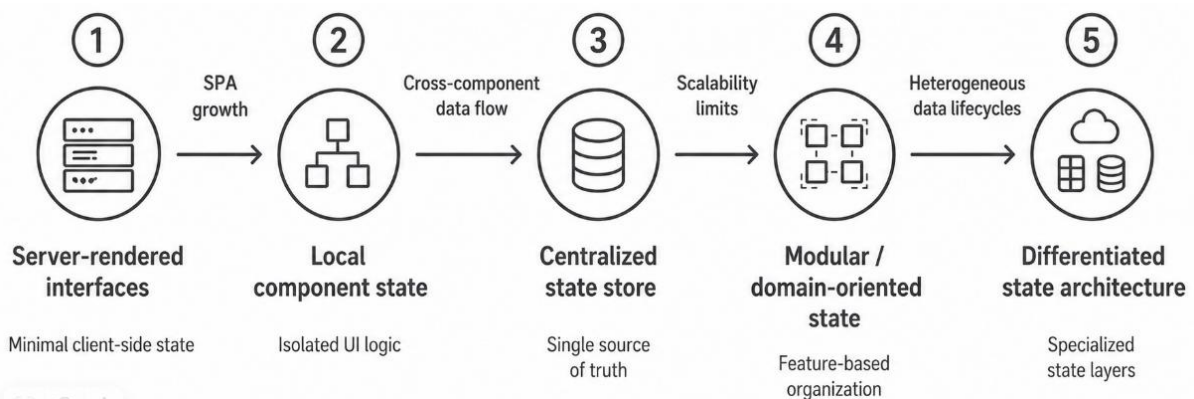


Fig. 1. Evolution of State Management Models in Frontend Architecture

The evolution of state management in frontend development can be described as a transition from **server-rendered interfaces** to differentiated state architecture. At the stage of server-rendered interfaces, most data and application logic were generated on the server, while the client layer was mainly responsible for page rendering, simple event handling, and local DOM updates; therefore, client-side state remained minimal and did not require complex synchronization.

With the growth of single-page applications, the role of **local component state** increased: routing, form validation, loading states, caching, error handling, and API interaction gradually moved to the frontend layer. This model

remains effective for isolated UI logic, but becomes limited when the same data must be shared between several components, since cross-component data flow increases coupling, complicates reuse, and creates a risk of duplication. These limitations led to the development of the **centralized state store**, where a shared store and formalized update mechanisms make state changes more reproducible, improve debugging, and support the single source of truth principle.

However, centralized stores also revealed scalability limits, including excessive boilerplate, complex dependencies, and over-centralization of temporary UI values. As a result, **modular/domain-oriented state** became more common: state is grouped by features, business entities, or data type,

which supports feature-based organization and reduces coupling between application areas. The current stage is associated with **differentiated state architecture**, where local hooks, domain stores, query caches, form tools, and derived selectors are combined according to heterogeneous data lifecycles. Modern state management is therefore not the choice of one library, but a system for distributing responsibility between application layers.

IV. CLASSIFICATION OF STATE MANAGEMENT MODELS IN MODERN FRONTEND APPLICATIONS

Modern frontend development does not rely on a single universal state management model. Several architectural models coexist, depending on application complexity, data lifecycle, and interaction patterns between components. The choice of a model is determined by whether the state is local or shared, temporary or persistent, synchronous or asynchronous, and client-derived or synchronized with a remote source. Therefore, state management models should be analyzed as architectural patterns that define how data are stored, updated, propagated, and tested within a frontend application (**Table 1**).

Table 1. Main State Management Models in Modern Frontend Applications

State management model	Core principle	Suitable use cases	Main limitation
Local component state	State remains inside one component or UI block.	Modal windows, tabs, temporary input values.	Poorly suited for shared state.
Lifted state	State is moved to the nearest common parent.	Small groups of related components.	Can lead to prop drilling.
Context-based state	Shared state is provided through a context layer.	Themes, authentication flags, interface modes.	Frequent updates may cause re-renders.
Centralized store	State is kept in a controlled shared store.	Large applications with complex data flows.	May create boilerplate and store overload.
Modular or feature-based stores	State is divided by feature or domain.	Independent modules and business scenarios.	Requires clear ownership rules.
Server-state model	Remote data are managed through cache and synchronization logic.	API responses, mutations, background refetching.	Depends on correct invalidation rules.
Form-state model	Form values and validation are managed separately.	Complex forms, field errors, submit status.	Requires coordination with domain/server state.
Hybrid model	Different mechanisms are combined by state type.	Complex systems with mixed state categories.	Requires architectural discipline.

This classification shows that state management is not reducible to the choice of one library. Local state is suitable for isolated interface behavior, while centralized and modular stores are used for shared domain logic. Server-state models manage remote data, cache synchronization, and asynchronous updates, whereas form-state models

handle input, validation, and submission logic. Therefore, mature frontend architecture usually develops as a hybrid model, where each state category is assigned to a mechanism that matches its scope, update frequency, and synchronization requirements.

V. STRUCTURING CLIENT-SIDE STATE IN FRONTEND APPLICATIONS

After identifying the main state management models, it is necessary to clarify what types of client-side state these models operate with. The transition from a universal global store to a differentiated state management architecture requires an explicit classification of state by its origin, scope, lifecycle, and synchronization requirements. In modern frontend systems, state should not be treated as a single set of variables shared across interface components. A more robust approach is to separate local UI state, domain

entities, server state, form state, navigation parameters, and derived values. This classification reduces component coupling, prevents unnecessary duplication, and allows each type of state to be handled by the mechanism that best corresponds to its technical role (**Table 2**).

Table 2. Main Categories of Client-Side State and Recommended Structuring Principles [2, 3]

State category	Typical examples	Recommended storage level	Engineering rationale
Local UI state	Menu expansion, active tabs, modal visibility, focus state, hover scenarios, temporary input values.	Component or small functional block.	Should remain close to the place of use, since it has no independent business value and does not require global synchronization.
Shared UI state	Interface mode, selected panel, shared filters, cross-component visibility flags.	Nearest common parent or lightweight shared context.	Should be lifted only when several independent interface areas depend on the same value.
Domain state	Users, orders, products, payments, roles, permissions, operation statuses.	Domain-oriented store or feature-level module.	Requires normalization, stable identifiers, selectors, and clear ownership by business domain.
Server state	API responses, cached collections, remote entities, request status, stale data.	Data-fetching layer, query cache, server-state tool.	Has an external source of truth and requires caching, invalidation, refetching, retry logic, and synchronization with the backend.
Form state	Field values, validation errors, touched fields, submit status, server-side errors.	Form-level state or specialized form library.	Has a short lifecycle and requires validation, reset logic, disabled states, and error processing.
Navigation state	Route parameters, query strings, pagination, sorting, selected route context.	Router, URL parameters, route-level state.	Should be reproducible from the navigation layer and remain compatible with deep links and browser history.
Derived state	Filtered lists, calculated totals, computed flags, aggregated values.	Selectors, memoized functions, computed values.	Should not be duplicated as independent store data if it can be derived from primary state.

Such structuring shows that frontend stability is achieved not through maximum centralization, but through a clear distribution of responsibility between state layers. Local UI state should remain close to the component, domain state

should rely on stable data models and selectors, and server state should be managed through caching, invalidation, refetching, optimistic updates, and rollback. Therefore, state management becomes an architectural composition in which each state

category is matched with an appropriate storage, update, and synchronization mechanism.

VI. STATE SYNCHRONIZATION IN SYNCHRONOUS AND ASYNCHRONOUS USER SCENARIOS

State synchronization in frontend applications includes both synchronous updates within the client layer and asynchronous coordination with remote data sources. Synchronous synchronization is required when several components depend on the same local UI state, domain entity, navigation parameter, filter, or derived value. In such cases, the main engineering task is to prevent duplication, inconsistent computed values, and uncontrolled propagation of changes across the component tree. Asynchronous synchronization becomes necessary when user actions, network requests, route changes, cache

updates, and API responses occur in parallel. Therefore, the interface must reflect not only the presence of data, but also its freshness, loading status, errors, partial updates, and possible staleness. Since server state has an external source of truth and may change independently of the current session, frontend architecture should include revalidation, request cancellation, cache invalidation, background updates, and error recovery mechanisms. At the user-experience level, such predictability supports trust and repeated interaction with a digital service [4].

Both synchronous and asynchronous scenarios can lead to inconsistent interface behavior if state ownership, update order, and synchronization rules are not explicitly defined. The main problems of state synchronization in frontend applications are summarized in

Table 3.

Table 3. Typical State Synchronization Problems in Frontend Applications

Problem	Technical cause	Interface manifestation	Engineering handling method
Stale data	Cached data no longer corresponds to the server state.	The user sees an outdated list, status, or field value.	Cache invalidation, stale time, background refetching.
Request race condition	Several requests are executed in parallel and return in a different order.	An older response overwrites a newer result.	Request cancellation, request ID, AbortController, response relevance check.
Mutation error	The server rejects a change after an optimistic update.	The interface shows the action as completed, although it has not been saved.	Rollback, refetch, error display, request retry.
State duplication	The same data are stored in several stores or components.	Different parts of the interface display different values.	Data normalization, single source of truth, derived selectors.
Loss of context during navigation	The component is unmounted before the request is completed.	The request result is lost or the state is updated incorrectly.	Cleanup functions, request cancellation, storing the result in the server-state layer.
Inconsistency after logout/login	The cache is not cleared when the user changes.	A new user may see data from the previous session.	Store reset, query cache cleanup, scoped cache keys.
Partial data update	A mutation changes only part of an entity or collection.	One screen is updated, while another remains in the previous state.	Targeted invalidation, patch update, normalized cache.

While synchronous consistency is mainly supported through clear ownership rules, selectors, derived values, and a single source of truth, asynchronous synchronization requires additional mechanisms for query lifecycle management. Queries retrieve and cache data, while mutations change server-side state. After a successful

mutation, the client can update a specific cache entry or invalidate related queries and trigger refetching. In practice, this logic is implemented through cache tags in RTK Query [5] and through `invalidateQueries` in TanStack Query [6]. The main mechanisms for handling asynchronous scenarios are summarized in **Table 4**.

Table 4. Engineering Mechanisms for Asynchronous State Synchronization

Mechanism	Purpose	Typical use case	Limitations
Query key	Identifies a request and its parameters in the cache.	Product lists, user cards, filtered selections.	Errors in key structure may lead to incorrect cache reuse.
Cache invalidation	Marks data as stale and triggers resynchronization.	Order update, record deletion, status change.	Overly broad invalidation increases API load.
Background refetching	Reloads data in the background without blocking the interface.	Feed updates, task notifications, statuses.	May generate excessive network traffic.
Optimistic update	Immediately displays the expected result of an operation.	Like action, adding an item to a cart, sending a message.	Requires rollback logic in case of failure.
Rollback	Restores state after an unsuccessful mutation.	Form saving error, server rejection, access-rights conflict.	Not always possible without storing the previous snapshot.
Retry mechanism	Repeats a request after a temporary error.	Unstable network, temporary API unavailability.	Should not be applied without limits to payments or critical operations.
Request cancellation	Cancels an outdated or irrelevant request.	Fast search input, route change, filtering.	Requires correct cancellation handling in code.
Normalized cache	Stores entities by identifiers.	GraphQL clients, complex domain models.	Increases the complexity of data schema configuration.

Thus, state synchronization should be treated as a dedicated architectural layer rather than as a set of isolated component-level effects. A stable frontend system separates local UI state, domain state, and server state, while synchronous and asynchronous operations are managed through explicit ownership rules, lifecycles, cache rules, invalidation strategies, and recovery mechanisms. This reduces the risk of inconsistent interface states, limits unnecessary coupling between components, and makes user scenarios more predictable under changing network and data conditions.

VII. ENGINEERING CRITERIA FOR EVALUATING STATE MANAGEMENT MODELS

The choice of a state management model should be assessed not only by library popularity or

framework compatibility, but also by engineering criteria that reflect the long-term stability of a frontend system. In complex client-side applications, state management affects component dependencies, data flow transparency, scenario reproducibility, and team development. Since architectural optimization of digital systems is associated with lower operating costs and higher service efficiency, sustainable frontend architecture is significant from both technical and organizational perspectives [7]. Therefore, state management models should be evaluated by maintainability, testability, predictability, scalability, performance overhead, and integration with asynchronous data exchange (**Table 5**).

Table 5. Engineering Criteria for Evaluating State Management Models [8-10]

Criterion	Evaluation focus	Indicators of a robust model	Typical architectural risk
Maintainability	Clarity of state ownership and update logic.	Explicit state boundaries, transparent data flow, limited state duplication.	Overloaded global store or excessive state fragmentation.
Testability	Ability to test state logic separately from UI components.	Pure transformations, isolated selectors, testable query and mutation logic.	Business logic embedded directly in components.
Predictability	Reproducibility of interface behavior under the same user actions.	Controlled updates, stable query keys, explicit lifecycle of asynchronous operations.	Hidden side effects and inconsistent state transitions.
Scalability	Ability to support the growth of features, screens, and modules.	Feature-level stores, domain decomposition, clear data ownership.	Tight coupling between independent parts of the application.
Performance	Impact on re-rendering, memory use, and data recalculation.	Selective subscriptions, memoized selectors, absence of redundant duplicated state.	Excessive re-renders and storage of computed values as independent data.
Team development	Convenience of parallel work within a shared codebase.	Unified conventions, standardized store structure, predictable data access patterns.	Conflicting implementations and inconsistent state organization.
Observability	Ability to trace state changes and localize defects.	Debugging tools, action history, query status tracking, error logging.	Difficult diagnosis of defects related to application state.

A 2025 comparative study of Redux, Zustand, and Context API confirms that the choice of a state management model has measurable technical consequences. Sharma et al. compared these approaches in React applications using performance, scalability, developer experience, responsiveness, re-rendering efficiency, and maintainability as evaluation criteria [11]. The study reports that Redux remained widely used in complex React projects, with 59.6% developer adoption and use in 72% of large-scale applications, while Zustand was associated with a 66.7% satisfaction rate and was more often selected for medium-complexity projects with lower boilerplate requirements. The same comparison reported implementation-level effects of structured state management, including a 45-47% reduction in state-related bugs, a 38% improvement in development efficiency, 55% more efficient code maintenance, and 35% better test coverage when isolated stores were used. Context API remained suitable for small applications and relatively static global values, but

could become inefficient when state changed frequently because additional re-renders might occur. These findings support the conclusion that state management should be selected according to application complexity, update frequency, scalability requirements, and maintainability needs rather than by library popularity alone.

Thus, the engineering value of a state management model is determined by its ability to preserve architectural clarity as the functional complexity of an application increases. A technically mature frontend system does not rely on a single universal approach, but combines local state, domain stores, query caches, form state, and derived selectors according to the lifecycle and ownership area of each data category. In this case, state management becomes a measurable architectural factor that affects not only interface behavior, but also the long-term maintainability, testability, and development efficiency of the application.

VIII. CONCLUSION

State management in modern frontend applications is not an auxiliary mechanism, but an independent architectural layer that determines interface predictability, the stability of user scenarios, and the maintainability of client-side logic. The evolution of state management approaches demonstrates a transition from the local state of individual components to more differentiated models, in which UI state, domain data, server state, forms, routing, and derived values are treated separately.

For complex frontend systems, the most effective approach is a hybrid model with clear responsibility distribution between application layers. Local state should remain within components, domain entities should follow business boundaries, and server data should be managed through caching, invalidation, and resynchronization. This approach improves testability, supports team development, reduces data desynchronization, and shows that state management is not only a library-level solution, but an architectural factor affecting performance, observability, and long-term frontend stability.

REFERENCES

- [1] M. Droń and T.M. Szandała, Web application state management: A review of leading React frameworks, *IEEE Software*, 43(2), 2026, 112-118. <https://doi.org/10.1109/MS.2025.3621269>.
- [2] N.R. Karka, State management in large-scale React applications: A comprehensive analysis, *International Journal of Advanced Research in Engineering and Technology*, 16(2), 2025, 108-132. https://doi.org/10.34218/IJARET_16_02_007.
- [3] T. Elias, Evolution of client-side web application development in the age of autonomous intelligent agents, *International Journal of Advanced Engineering, Management and Science*, 12(3), 2026, 200-206. <https://doi.org/10.22161/ijaems.123.18>.
- [4] E. Lim, Formation of customer loyalty in the banking sector based on the implementation of intuitive digital interfaces, *Economy and Business: Theory and Practice*, 2(132), 2026, 145-151.
- [5] Redux Toolkit, Automated re-fetching, <https://redux-toolkit.js.org/rtk-query/usage/automated-refetching> (accessed 15 June 2026).
- [6] TanStack Query, QueryClient, <https://tanstack.com/query/v5/docs/reference/QueryClient> (accessed 15 June 2026).
- [7] K. Kovalevskyi, Impact of architectural optimization of distributed systems on reducing operating costs and improving the economic efficiency of digital services, *International Journal of Engineering in Computer Science*, 8(2), 2026, 63-67. <https://doi.org/10.33545/26633582.2026.v8.i2a.278>.
- [8] F. Antunes, M.J.D. Lima, M.A.P. Araújo, D. Taibi, and M. Kalinowski, Investigating benefits and limitations of migrating to a micro-frontends architecture, *Proc. 38th Brazilian Symposium on Software Engineering*, 2024, 103-113. <https://doi.org/10.5753/sbes.2024.3303>.
- [9] N. Kaushik, H. Kumar, and V. Raj, Micro frontend based performance improvement and prediction for microservices using machine learning, *Journal of Grid Computing*, 22, 2024, Article 44. <https://doi.org/10.1007/s10723-024-09760-8>.
- [10] H. Metin and D. Bork, A reference architecture for the development of GLSP-based web modeling tools, *Software and Systems Modeling*, 24, 2025, 1869-1895. <https://doi.org/10.1007/s10270-024-01257-y>.
- [11] N. Sharma, S. Charan, Shaan, and Suma, Performance and developer experience comparison of Redux, Zustand, and Context API in React applications, *International Journal on Science and Technology*, 16(2), 2025. <https://doi.org/10.71097/IJSAT.v16.i2.5026>.