# Architecting a Hybrid Vanilla JavaScript State Management Library: Leveraging Dependencies for Rapid Development and Robustness

## I. Executive Summary

### A. Project Vision and Hybrid Approach

The primary objective is the development of a comprehensive state management library using vanilla JavaScript. Acknowledging the imperative for rapid initial development and robust functionality from its first iteration, the strategy involves a hybrid approach. This entails leveraging the capabilities of established, battle-tested libraries—namely Killa.js, VanJS, Immutable.js, and Redux—as foundational dependencies. This methodology is designed to accelerate the release timeline while ensuring a feature-rich and reliable solution. The long-term vision, as outlined in prior dependency-free research [1], aims for a completely self-reliant library. Consequently, current architectural decisions will be made with this eventual transition in mind, incorporating core principles and patterns that facilitate future independence.

### B. Core Architectural Pillars

The proposed library will be built upon a dual-store architecture. This comprises a global store, designed for managing complex, application-wide state, and a scoped store mechanism, tailored for localized, component-level state management. Immutability is a non-negotiable, foundational principle, which will be rigorously enforced through the integration of Immutable.js.[2] A key tenet of the design is to augment and enhance, rather than supplant, the native state management capabilities inherent in Web Components and the Document Object Model (DOM).

### C. Key Benefits of the Proposed Solution

This hybrid strategy offers several distinct advantages. Firstly, it facilitates rapid initial development and a significantly shorter time-to-market for the library. Secondly, by amalgamating the strengths of multiple mature libraries, users will gain access to a rich and diverse feature set from the outset. Finally, this approach establishes a clear and extensible pathway for future enhancements, including the planned integration of middleware, developer tools, and state persistence mechanisms.

## II. Foundational Architecture: Core State Mechanisms

### A. Overview: Selecting and Integrating Dependencies

The decision to adopt a hybrid approach, utilizing existing libraries as dependencies,

stems from a pragmatic balance between the desire for a custom solution and the need for initial development velocity and reliability. Building entirely from scratch, while offering complete control, would significantly extend the development timeline and introduce risks associated with implementing complex state management primitives. Mature libraries, on theother hand, have undergone extensive testing and community vetting.

The selection of Killa.js, VanJS, Immutable.js, and Redux is deliberate, with each library contributing specific strengths that align with the project's multifaceted requirements. Killa.js offers a lightweight approach to state management and includes features like persistence.[3] VanJS provides an elegant solution for reactive, scoped state tightly coupled with direct DOM manipulation, ideal for UI components.[4] Immutable.js is the cornerstone for ensuring data integrity through persistent immutable data structures.[2] Redux offers a well-defined pattern for managing global application state, particularly its action/reducer paradigm, which is conducive to building extensible systems with middleware and developer tooling.[6]

The following table provides a comparative overview of how these libraries address key state management features and how the proposed hybrid library will synthesize these capabilities in its initial design.

**Table 1: Comparative Overview of Core State Management Features**

| Feature | Killa.js | VanJS | Immutable.js | Redux | Proposed Hybrid Library (Initial Design Focus) |
|---|---|---|---|---|---|
| **Global Store API - Creation** | createStore(initialState, options) [3] | N/A (Primarily scoped) | N/A (Data structures) | createStore(reducer, initialState, enhancer) [6] | createGlobalStore(rootReducer, initialState, enhancer) (Redux-inspired, using redux and immutable) |

| | | | | | |
|---|---|---|---|---|---|
| **Global Store API - Update** | store.setState(updaterFn) [3] | N/A | N/A | store.dispatch(action) with reducers [6] | store.dispatch(action) with reducers (Redux-inspired, reducers use immutable) |
| **Global Store API - Access** | store.getState() [3] | N/A | N/A | store.getState() [6] | store.getState() (Returns Immutable.js structure) |
| **Global Store API - Subscription** | store.subscribe(listener,?selector) [3] | N/A | N/A | store.subscribe(listener) [6] | store.subscribe(listener) (Redux-inspired) |
| **Scoped State API - Creation** | N/A | van.state(initVal) [4] | N/A | N/A (Global focus) | createScopedState(initialValue) (VanJS-based, using vanjs) |
| **Scoped State API - Update** | N/A | state.val = newValue [4] | N/A | N/A | scopedState.val = newValue (VanJS-based) |
| **Scoped State API - Access** | N/A | state.val, state.oldVal, state.rawVal [4] | N/A | N/A | scopedState.val, scopedState.oldVal, scopedState.rawVal (VanJS-based) |
| **Scoped State API - Binding** | N/A | Direct DOM property/child binding [4] | N/A | N/A | Direct DOM binding (Leveraging |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | VanJS capabilities) |
| **Immutability Approach** | Not inherently enforced by core; relies on developer discipline or middleware. | Not inherently enforced; state.val can be mutable. | Persistent data structures (Map, List, etc.) [2] | Relies on reducers returning new state; often paired with immutable libraries. [7] | Mandatory for global store (via Immutable.js); optional but recommended for complex scoped states. |
| **Middleware Support (Initial)** | persist middleware available [3] | N/A | N/A | applyMiddleware for extensive middleware ecosystem [8] | Redux-style applyMiddleware for global store. |
| **Persistence Support (Initial)** | persist middleware (e.g., localStorage) [3] | N/A | N/A | Via middleware (e.g., redux-persist). | Killa-inspired persistence middleware for global store. |
| **DevTools Integration (Potential)** | Not explicitly mentioned. | Not explicitly mentioned. | N/A | Strong integration with Redux DevTools [11] | Plan for Redux DevTools integration for global store. |

This comparative analysis underscores the rationale for selecting specific features from each library. The goal is not merely to aggregate these libraries but to create a cohesive system where their strengths are synergistic, addressing the diverse state management needs of modern web applications. The "Proposed Hybrid Library" column reflects the initial design choices, aiming for a powerful yet pragmatic solution.

### B. Designing the Global Store (Drawing from Redux & Killa.js)

#### Core Philosophy: Centralized and Predictable State

The global store will adhere to Redux's principle of a single source of truth, providing a centralized repository for the application's entire state tree.[6] This centralization simplifies state management by making data flow predictable and changes traceable, which is invaluable for debugging and maintaining complex applications. While Killa.js offers a more lightweight API [3], the structured action/reducer paradigm of Redux is favored for the global store. This choice is strategic: the explicit flow of actions processed by pure reducers provides a more robust foundation for the planned future integration of middleware and developer tools, which often rely on intercepting and understanding these discrete state transitions.[10]

**API for Global Store Management**

The Application Programming Interface (API) for the global store will be heavily inspired by Redux, ensuring familiarity for developers accustomed to its patterns, while also incorporating considerations for immutability from the outset.

- Store Creation:
  The primary method for creating a global store will be createGlobalStore(rootReducer, initialState, enhancer). This mirrors Redux's createStore function.6
  - rootReducer: A function that takes the current state and an action, and returns the new state. This reducer will be composed of smaller reducers, each managing a slice of the state, likely using a utility similar to Redux's combineReducers.[14] If the entire global state is an Immutable.Map, redux-immutable's version of combineReducers would be appropriate.[16]
  - initialState: An optional argument to preload the store with a previously existing state, for example, when rehydrating from persistent storage. This state should be an Immutable.js data structure.
  - enhancer: An optional argument for extending the store's capabilities, primarily for applying middleware using a function like applyMiddleware.[8]
- State Retrieval:
  The method store.getState() will return the current state tree of the application.3 Crucially, the object returned by getState() will be an Immutable.js data structure (e.g., Immutable.Map), ensuring that consumers cannot accidentally mutate the state directly.
- State Updates (Actions & Reducers):
  State modifications will be exclusively managed through the dispatch of actions, a core tenet of Redux.6
  - store.dispatch(action): This is the sole mechanism for initiating a state change. An action is a plain JavaScript object that must have a type property

(typically a string constant) describing the intended change. It can also carry a payload with any data necessary for the update.
    - **Reducers**: These are pure functions with the signature (currentState, action) => newState.[6] Each reducer is responsible for calculating the next state slice based on the current state slice and the dispatched action. Reducers *must not* mutate the currentState; instead, they must use Immutable.js methods (e.g., .set(), .updateIn(), .merge()) to produce and return a new Immutable.js instance representing the updated state.[2] While Killa.js provides a store.setState(updaterFn) method [3], which is simpler for direct state updates, the Redux action/reducer pattern offers superior traceability, testability, and extensibility, especially concerning middleware. Killa's concept of defining internal actions using get and set within createStore [3] can serve as an inspiration for structuring action creator functions that encapsulate the logic for creating action objects.
- Subscriptions:
  To react to state changes, the store will provide a subscription mechanism:
    - store.subscribe(listener): This method registers a listener callback function that will be invoked whenever an action is dispatched and the state tree may have changed.[3] The listener function itself does not receive the state as an argument; instead, it should call store.getState() to access the latest state.
    - The subscribe method will return an unsubscribe function, which, when called, removes the listener from the store.
- State Reset/Overwrite:
  While Killa.js offers a direct store.resetState(newState?) method 3, such functionality in this library will be implemented through a dedicated action type (e.g., { type: 'RESET_GLOBAL_STATE', payload: newGlobalState }). This approach aligns with the Redux pattern of all state changes being traceable to a dispatched action, maintaining consistency and predictability.

**Integrating Immutability.js for Global State Integrity**

The integrity of the global state will be guaranteed by Immutable.js.[2] The entire global state tree, as well as each individual slice managed by a reducer, will be an Immutable.js collection (e.g., Immutable.Map for object-like structures and Immutable.List for array-like structures).

Reducers are central to this strategy. They will be designed to receive an Immutable.js structure as their current state and must return a new Immutable.js structure representing the next state. Operations within reducers will exclusively use Immutable.js APIs such as set(), getIn(), update(), merge(), push(), etc..[2] This practice

is not merely a convention but a requirement for leveraging Redux's change detection mechanisms, which often rely on shallow equality checks (comparing object references) to determine if a part of the state has changed.[9] If a reducer mutates state directly, these checks would fail, leading to components not re-rendering correctly. Furthermore, immutability is a prerequisite for advanced debugging features like time-travel debugging, as it ensures that each historical state is preserved accurately.[9]

The use of redux-immutable can significantly simplify the enforcement of these principles. Its combineReducers utility is specifically designed to work with an Immutable.js state tree, ensuring that the overall state is an Immutable.Map and that individual slice reducers correctly handle and return Immutable.js objects.[16] This combination of Redux for flow control and Immutable.js for data integrity, potentially streamlined by redux-immutable, forms a robust and well-established pattern for managing global application state.

### C. Implementing Scoped State (Leveraging VanJS)

#### Core Philosophy: Lightweight, Reactive, and UI-Bound

For managing state that is localized to specific UI components, particularly custom web elements, a different approach is warranted. VanJS, with its focus on lightweight, reactive state and direct DOM manipulation without an intermediary virtual DOM, provides an ideal foundation.[4] The philosophy here is to offer a state mechanism that is closely tied to the UI elements it controls, allowing for efficient and direct updates. This aligns with the requirement to augment native DOM capabilities rather than replacing them [User Query]. VanJS's design is well-suited for scenarios where numerous small, independent stateful components exist, as its minimal overhead is advantageous compared to instantiating a full Redux-like store for each.[22]

#### API for Scoped State Management

The API for scoped state will primarily wrap and expose the core functionalities of VanJS's state objects, making them accessible within the ecosystem of the new library.

- **State Creation:**
  - createScopedState(initialValue): This function will serve as a lightweight wrapper around VanJS's van.state(initialValue).[4] It will return a state object compatible with VanJS's reactivity system. The initialValue is typically a primitive value (string, number, boolean) or a simple JavaScript object/array. For more complex scoped states, initialValue could also be an Immutable.js

data structure, though this is not enforced at this level.

- State Access/Update:
  The returned scopedState object will expose properties for interaction, mirroring VanJS:
  - scopedState.val: This property allows both reading the current value of the state and updating it. Assigning a new value to scopedState.val will trigger reactive updates in any DOM elements bound to this state.[4]
  - scopedState.oldVal: A read-only property providing access to the previous value of the state before the latest update, useful for certain UI logic or transitions.[5]
  - scopedState.rawVal: A read-only property to access the current value without establishing a reactive dependency, useful in specific scenarios within derived state computations.[5]
- **Derived State:**
  - deriveScopedState(derivationFn): This function wraps van.derive(derivationFn).[5] The derivationFn is a function that computes a new value based on one or more other state objects (which can be other scoped states or, via the bridging mechanism, global state). The returned derived state will automatically update whenever its dependencies change.
- Binding to DOM Elements:
  A core strength of VanJS is its direct and efficient binding of state to DOM elements.4 The library will leverage this by encouraging patterns where VanJS tag functions or van.add are used to create DOM structures whose properties or child content are reactively tied to scopedState objects. For instance, within a custom web component, its internal DOM can be built using VanJS tags, with attributes and text nodes directly linked to scoped states initialized by the component. The library might offer helper functions to simplify common binding scenarios, such as component.bindProperty(element, 'value', scopedState) or component.bindChild(parentElement, scopedState), which would internally use VanJS's binding mechanisms. VanJS's attr() utility within Van Elements provides a clear pattern for binding attributes to state.23

**Encapsulation and Isolation**

Scoped states are intended to be encapsulated within the component or module that creates them. When createScopedState is called within, for example, a custom element's class definition, the resulting state object is local to that instance of the element. This natural isolation is a key feature of VanJS's typical usage pattern and prevents unintended state interference between different components or parts of the application.[5] This directness and focus on real DOM elements make VanJS an

excellent fit for augmenting the capabilities of standard Web Components, which also operate with their own encapsulated DOM (Shadow DOM) and lifecycle methods.[22]

### D. Bridging Scoped and Global State

#### Requirement and Interaction Patterns

A critical requirement is to enable "seamless access to the global scope from the isolated scope" [User Query]. This implies that UI components managing their local view state with scoped states must also be ableto read from and, where appropriate, initiate changes to the global application state. However, this access must be managed to prevent tight coupling and maintain the predictability of the global state. Direct, unrestricted exposure of the global store instance to every scoped context is generally undesirable. Instead, a more controlled interface is proposed, drawing inspiration from how libraries like React-Redux manage connections between components and the store.[25]

The interaction will be facilitated through the following patterns:

- Read-Only Access to Global State:
  Scoped instances can retrieve data from the global store. This will be achieved by providing the scoped context with a way to execute selector functions against the global state.
  - scopedContext.getGlobalState(selectorFn): This method, available on a connected scoped state's context, would take a selectorFn. The selectorFn is a function that receives the global Immutable.js state object and returns a specific piece of data or a derived value. This ensures that the scoped component only accesses the data it needs.
- Subscribing to Global State Changes:
  Scoped components often need to react to changes in specific parts of the global state.
  - scopedContext.subscribeToGlobal(selectorFn, callback): This method allows a scoped instance to register a callback function that is executed only when the data selected by selectorFn from the global state changes. Internally, this would involve subscribing to the global store and, upon each global update, re-running the selectorFn and comparing its previous result with the new result (using Immutable.is() for immutable data or === for primitives) before invoking the callback. This targeted subscription mechanism prevents unnecessary updates in the scoped component if irrelevant parts of the global state change.
- Dispatching Actions to Global Store:

Scoped components may need to trigger changes in the global application state.

- scopedContext.dispatchGlobal(action): This method provides a way for the scoped context to send an action object to the global store's dispatch queue. This maintains the Redux pattern where all global state changes are initiated via explicit actions.

**Implementation Strategy**

The connection between a scoped state and the global store will be established explicitly. When a scoped state is created (e.g., via createScopedState) within a component that requires global interaction, an instance of the global store (or a facade providing the necessary getState and dispatch methods) can be passed as an option:
const myScopedState = createScopedState(0, { globalStore });
The createScopedState function would then augment the returned scoped state object or its context with the getGlobalState, subscribeToGlobal, and dispatchGlobal methods. This connection should be optional; if no globalStore is provided, the scoped state remains fully isolated and these bridging methods would either be unavailable or no-ops.

This explicit "connection" step ensures that the dependency on the global store is clear and managed. The provision of specific interaction methods, rather than exposing the entire global store object, defines a clearer contract and promotes better encapsulation. If numerous scoped states subscribe to the global store, performance considerations are important. The subscribeToGlobal mechanism must be efficient, ensuring callbacks are invoked only when truly necessary. This involves careful comparison of selected state slices, leveraging the benefits of immutability for efficient change detection.

**E. Immutability Strategy (with Immutable.js as a Dependency)**

**Core Dependency and Key Data Structures**

Immutable.js will serve as the foundational library for ensuring state immutability across the application, particularly within the global store.[2] Its persistent data structures are key to this strategy:

- **Immutable.Map**: This will be the primary structure for representing object-like states, including the root of the global state tree and individual state slices managed by reducers.
- **Immutable.List**: This will be used for array-like data within the state, ensuring that operations on lists also result in new, immutable lists.
- Immutable.Record could be considered for more structured, predefined object shapes in advanced scenarios, offering a balance between the flexibility of Map

and the defined fields of a class, though it's not a primary focus for the initial implementation.

### Core Operations

The library will rely on core Immutable.js operations for all state manipulations within the global store and for any scoped states that opt into using immutable structures:

- **Creation/Conversion:** Immutable.fromJS() is essential for converting plain JavaScript objects and arrays (often received from API responses or used as initial state definitions) into their deeply immutable counterparts.[2] This ensures that all data entering the managed state becomes immutable.
- **Reading:** Data will be accessed using get(key) for top-level properties in a Map or List, and getIn([path]) for deeply nested values.[2] These methods provide safe access without the risk of accidental mutation.
- **Updating:** All update operations, such as set(key, value), setIn([path], value), update(key, updaterFn), updateIn([path], updaterFn), merge(), and list-specific methods like push(), pop(), splice(), etc., are designed to return new immutable instances.[2] The original data structure remains unchanged, which is the essence of immutability.

### Enforcement and Usage

- **Global Store:** Immutability is mandatory for the global store. Reducers interacting with the global store *must* be written to expect Immutable.js data structures as input for their state slice and *must* return new Immutable.js data structures. If redux-immutable is integrated, its specialized combineReducers function will help enforce this, ensuring that the entire state tree managed by Redux is composed of Immutable.js collections.[16]
- **Scoped State (Optional but Recommended):** For VanJS-based scoped states, the use of Immutable.js is not strictly enforced by VanJS itself, as van.state().val can hold any JavaScript value. However, the library will strongly recommend and facilitate the use of Immutable.js for complex objects or arrays within scoped states. This can be achieved by initializing a scoped state with an immutable structure: const immutableScopedState = createScopedState(Immutable.Map({ count: 0 })); Subsequent updates would then involve retrieving the current immutable value, performing an immutable operation, and setting the new immutable structure back to scopedState.val: immutableScopedState.val = immutableScopedState.val.set('count', 1); This approach offers a balance: the global store benefits from guaranteed immutability, while simpler scoped states (e.g., a boolean toggle) can avoid the overhead of Immutable.js if not needed.

However, for consistency and to prevent subtle bugs when dealing with objects or arrays in scoped state, encouraging immutability is a good practice.

### Performance Benefits and Structural Sharing

A significant advantage of Immutable.js is its use of structural sharing.[2] When a new immutable collection is created from an existing one, Immutable.js reuses the underlying data nodes that haven't changed. Only the modified parts and their ancestors in the data tree are newly allocated. This makes immutable operations highly efficient, especially for large data structures, as it avoids deep copying entire objects. This efficiency is critical for Redux's performance, as it enables quick reference equality checks (===) to determine if a state slice has actually changed, thereby optimizing re-renders.[2]

### Interoperability

While working with immutable structures internally is beneficial, interaction with other parts of the system (e.g., UI rendering libraries that expect plain objects, serialization for network requests or localStorage) may require conversion. Immutable.js provides the toJS() method for deeply converting immutable collections back to plain JavaScript objects and arrays.[2] However, toJS() can be a performance-intensive operation, especially for large or deeply nested structures, so its use should be judicious and typically localized to the boundaries of the system.

The developer experience when working directly with Immutable.js APIs can sometimes be more verbose than manipulating plain JavaScript objects (e.g., state.getIn(['path', 'to', 'value']) versus state.path.to.value). To mitigate this, the library could provide a thin layer of utility functions that simplify common immutable update patterns, especially within reducers. For example, a helper like updateInGlobalState(['path', 'to', 'value'], newValue) could abstract the underlying store.dispatch and Immutable.js setIn logic. This not only improves developer ergonomics but also creates an abstraction layer that could be reimplemented without Immutable.js in a future dependency-free version of the library, aligning with the project's long-term goals.

## III. DOM and Event State Management Strategy

### A. Augmenting Native Web Component State

The library's philosophy is to enhance, not replace, the inherent state management capabilities of Web Components. Web Components possess their own lifecycle methods, attribute/property systems, and often an encapsulated Shadow DOM.[22] The

state management solution should integrate smoothly with these native features.

- Web Component Lifecycle Integration:
  Scoped states, being VanJS-based, are well-suited for Web Components. They can be initialized within the component's constructor or, more commonly, in the connectedCallback() lifecycle method, which is invoked when the component is added to the DOM.24 This ensures the state is ready when the component becomes active. Conversely, any subscriptions established by the component to either its own scoped states (if using VanJS's onnew or van.derive for side effects) or to the global store must be meticulously cleaned up in the disconnectedCallback() method.24 This prevents memory leaks when the component is removed from the DOM.
- Reflecting State to Attributes/Properties:
  VanJS's reactive state binding capabilities can be directly used to reflect the internal scoped state of a Web Component to its public attributes or properties.4 For example, a custom button component might have an internal scoped state isLoading = createScopedState(false). This state can be bound to the component's disabled property and potentially a visual loading indicator within its Shadow DOM. Van Element's attr() utility provides a declarative way to link attributes to VanJS states 23, a pattern that can be adopted or emulated.
- Responding to Attribute/Property Changes:
  Web Components can react to changes in their attributes through the attributeChangedCallback(name, oldValue, newValue) lifecycle method, provided the attributes are observed via a static observedAttributes getter.24 This callback can be used to synchronize an internal scoped state if an attribute is modified externally. For instance, if a parent element changes an attribute on the custom element, attributeChangedCallback can update the corresponding scoped state, ensuring the component's internal logic and UI react accordingly.

This synergy between VanJS-based scoped states and standard Web Component practices is a key strength. Because VanJS operates directly on real DOM elements [4], it avoids the potential impedance mismatch that can arise when trying to integrate Virtual DOM-based libraries with the Web Component model.

**B. Patterns for Managing UI-Specific States (Focus, Scroll, Input Values, etc.)**

Many UI interactions involve transient states that are best managed locally.

- Local Scoped State for UI Elements: For common UI states such as the current value of an input field, the hover state of a button, or the expanded/collapsed status of an accordion panel, VanJS-based scoped states offer a lightweight and

efficient solution.4
For example: const inputValue = createScopedState(''); can be bound to an <input> element: van.tags.input({type: 'text', value: inputValue, oninput: e => inputValue.val = e.target.value}).

- Focus Management:
  Managing which element currently has focus, or should receive focus programmatically, can be handled through state. While document.activeElement provides the currently focused element 28, a declarative approach involves storing an identifier (e.g., an ID or a unique key) of the target element in a state variable (either scoped or global, depending on the context). A van.derive or a store subscription can then observe this state. When the state changes to a new target identifier, a side effect can call the .focus() method on the corresponding DOM element.29 The preventScroll option (element.focus({preventScroll: true})) should be considered if the default browser scroll-into-view behavior is not desired and needs to be controlled by application logic.29
- Scroll Position:
  Similar to focus, the scroll positions of specific elements or containers can be stored in state if the application needs to programmatically control or restore them. Event listeners for the scroll event on these elements would update the state with the current scroll coordinates (scrollTop, scrollLeft). Conversely, changes to this state (e.g., a user action to jump to a section) could trigger imperative calls to element.scrollTo() or element.scrollLeft/element.scrollTop.
- DOM Event State:
  The requirement to manage "DOM and Event states" [User Query] implies that certain aspects of DOM events themselves might need to be captured and stored if they influence application logic beyond the immediate event handler. For example, tracking mouse coordinates during a drag operation, or accumulating a sequence of key presses, could involve updating state variables. These state variables can then be used by other parts of the application or by derived states to trigger further logic or UI updates.30

**C. Declarative Event Handling Linked to State**

The library should promote a declarative style for event handling, where UI events directly influence state, and state changes declaratively update the UI.

- VanJS Native Declarative Event Handling:
  VanJS inherently supports a declarative event handling model. Event handlers in VanJS tag functions are typically simple functions that directly modify VanJS state objects.4 For example:

van.tags.button({onclick: () => counterState.val++}, 'Increment')
This direct linkage between UI events and state updates is a core aspect of VanJS's reactivity.

- Global Event Listeners Conditional on State:
  There are scenarios where event listeners need to be attached to global objects like window or document (e.g., for global keyboard shortcuts, online/offline status detection, or closing a modal on an outside click). The attachment or detachment of these listeners might depend on the application's state (e.g., a modal being visible, a specific user mode being active).
  This can be managed reactively. A van.derive (for scoped state dependencies) or a global store subscription can monitor a boolean state variable that controls whether a global listener should be active. When this control state changes, the derive function or subscription callback can execute window.addEventListener(...) or window.removeEventListener(...) accordingly. Some libraries offer declarative ways to manage global event listeners based on component lifecycle or state 32, and similar utilities could be provided by this library. For example, a helper function manageGlobalListener(target, eventName, handlerFn, activeState) could encapsulate this logic, where activeState is a boolean state object from the library.

While VanJS excels at declarative state-to-DOM binding, there might be instances involving highly complex DOM interactions (e.g., intricate animations driven by physics, integration with third-party libraries that manipulate the DOM directly) that are difficult to express purely declaratively. Since VanJS works with real DOM elements, these elements are accessible for imperative manipulation if necessary.[4] The library should provide guidance on balancing VanJS's declarative paradigm with these occasional imperative escape hatches, ensuring that direct DOM manipulations do not conflict with VanJS's reactive updates. This might involve patterns for obtaining stable references to DOM elements managed by VanJS, akin to how refs are used in React for direct DOM access.[34]

## IV. Initial Implementation and Distribution (ES6 Modules)

### A. Proposed Module Structure

A clear and logical module structure is essential for maintainability, scalability, and the eventual transition to a dependency-free library. The initial structure will encapsulate functionalities and their primary dependencies.

- **Core Modules:**
  - src/index.js: The main entry point for the library, exporting the public API. This

module will re-export selected functionalities from other core modules.
    - src/globalStore.js: Contains the implementation for the global store, including createGlobalStore and related utilities. This module will primarily depend on redux and immutable.
    - src/scopedState.js: Houses the logic for scoped state management, including createScopedState and deriveScopedState. Its main dependency will be vanjs-core.
    - src/bridge.js: Provides the utilities and mechanisms for enabling communication and data flow between scoped states and the global store (e.g., connectToGlobal or methods attached to scoped states when connected). This module will interact with both globalStore and scopedState concepts.
    - src/constants.js: (Optional but recommended) Defines common string constants, such as standard action types (e.g., for state reset), to avoid magic strings.
- **Utilities:**
    - src/utils/immutableUtils.js: A collection of helper functions for working with Immutable.js data structures. This could include wrappers for common update patterns (e.g., setInIfNotEqual, pushUniqueToList) or a utility like ensureImmutable(data) that converts data to an Immutable.js structure if it isn't already. This promotes consistency and can simplify reducer logic.
    - src/utils/domUtils.js: (Potential) Helpers for common DOM interactions, especially those related to binding state or managing event listeners in a way that complements VanJS.

All internal code and public exports will strictly adhere to ES6 module syntax (import and export), aligning with the preference for direct browser consumption and seamless integration with modern JavaScript build toolchains [User Query]. This modular approach, by isolating dependencies, is a critical step. For example, globalStore.js internally utilizes redux, but its exported API should be designed to be as generic as possible. If, in the future, redux is replaced with a custom implementation, the changes would ideally be confined to globalStore.js, leaving the public API and consuming application code unaffected, provided the facade remains stable.

### B. Browser-First Development and Future Build Considerations

The initial development phase will prioritize modern browser environments that natively support ES6 modules.

- **Initial Target Environment:** Modern evergreen browsers (Chrome, Firefox, Safari,

Edge) with robust ES6 module support.

- **Development Setup:** A lightweight development server that handles ES6 modules efficiently, such as Vite or live-server with appropriate configurations, is recommended. This allows for rapid iteration without complex build configurations during early development.
- Future Build Step for Distribution:
  As the library matures and aims for broader adoption, a dedicated build process will be necessary. This process will transform the ES6 module-based source code into various distribution formats suitable for different consumption environments:
  - **ESM (ES Modules):** For modern bundlers and browsers.
  - **UMD (Universal Module Definition):** For compatibility with older systems, direct <script> tag usage, and some Node.js environments.
  - **CJS (CommonJS):** Primarily for Node.js environments. Tools like Rollup or esbuild are well-suited for this task. Killa.js, for instance, utilizes esbuild.config.js, indicating a similar modern and efficient build strategy.[3]

  A strong consideration, inspired by Killa.js's significant adoption of TypeScript (93.3% of its codebase [3]), is the use of **TypeScript for internal development**. While the user query specifies vanilla JavaScript as the output and for early adopters, developing the library in TypeScript offers substantial benefits:
  - **Improved Code Quality and Maintainability:** Static typing helps catch errors during development, refactor with greater confidence, and makes the codebase easier to understand and navigate.
  - **Enhanced Developer Experience:** Better autocompletion, type checking, and inline documentation in IDEs.
  - **High-Quality Type Definitions:** TypeScript allows for the generation of accurate .d.ts declaration files, which are invaluable for developers using the library in their own TypeScript projects, even if the distributed library code is plain JavaScript. The TypeScript code would be transpiled to clean, idiomatic ES6 JavaScript for distribution. This approach combines the robustness of TypeScript development with the accessibility of vanilla JS for end-users.

## V. Roadmap for Advanced Features (Future Considerations)

### A. Middleware Architecture (Inspired by Killa/Redux)

A middleware layer is a crucial extensibility point for a state management library, allowing developers to inject custom logic into the dispatch pipeline of the global store. This is commonly used for handling side effects like asynchronous API calls, logging state changes and actions, implementing custom action transformations, or integrating routing.

- Concept and Inspiration:
  The middleware will intercept actions dispatched to the global store before they reach the reducers.3 Both Redux and Killa.js provide mechanisms for middleware. Redux's applyMiddleware enhancer and its standard middleware signature are particularly influential due to their widespread adoption and composability.10 Killa.js demonstrates middleware usage with its persist functionality, applied via a use array in the createStore options.3
- **Proposed API:**
  - **applyMiddleware(...middlewares):** This function will be provided as an enhancer to createGlobalStore. It takes one or more middleware functions as arguments and returns a function that enhances the store's dispatch method. This is identical to the Redux pattern.[8]
  - **Middleware Signature:** The standard Redux middleware signature will be adopted for maximum compatibility and familiarity: const myMiddleware = store => next => action => { // middleware logic here return next(action); }; [10]
    - store: An object providing a limited store API, typically { getState, dispatch }. This dispatch allows middleware to dispatch new actions (e.g., for asynchronous operations).
    - next: A function that, when called with an action, passes the action to the next middleware in the chain. If this is the last middleware, next will be the original store.dispatch that sends the action to the reducers.
    - action: The currently dispatched action object. The middleware must eventually call next(action) (or a new action) to continue the dispatch process, unless it intends to halt the action entirely.
- Integration:
  The applyMiddleware function works by composing the middlewares into a chain. The store's dispatch function is replaced by this chain, so when store.dispatch(action) is called, the action flows through each middleware before reaching the reducers. This composability is a powerful feature, allowing for complex sequences of operations to be built from smaller, focused middleware functions.
- Initial Use Cases:
  Common initial middleware could include:
  - **Logging:** To log actions and state changes to the console for debugging.
  - **Asynchronous Actions (Thunk-like):** A simple middleware to allow action creators to return functions (thunks) instead of action objects. These thunks receive dispatch and getState as arguments, enabling them to perform asynchronous operations and dispatch further actions upon completion.

While Killa.js's use: [persist(...)] [3] is straightforward for applying pre-defined middlewares, the Redux applyMiddleware pattern offers greater flexibility for user-defined and third-party middlewares, aligning better with the goal of an "all-encompassing" solution.

**B. Developer Tools Integration (Inspired by Redux DevTools)**

Providing robust debugging capabilities is essential for a modern state management library. Integrating with the existing Redux DevTools browser extension offers a feature-rich debugging experience with minimal development overhead for the library itself.

- Mechanism and API:
  The integration will be achieved by communicating with the Redux DevTools extension through the API it exposes on the window object, typically window.__REDUX_DEVTOOLS_EXTENSION__.11 This API allows custom stores to connect to the DevTools, send state and action information, and respond to DevTools commands like time-travel.
- **Core window.__REDUX_DEVTOOLS_EXTENSION__ Methods to Utilize:**
  - **devTools = window.__REDUX_DEVTOOLS_EXTENSION__.connect(config)**: This method establishes a connection with the DevTools extension. The config object can pass options like the instance name (useful if multiple stores are being debugged on one page) and specify supported features (e.g., jump, skip). [36]
  - **devTools.init(initialGlobalState)**: Called once after connecting, this sends the initial state of the global store to the DevTools, allowing it to display the starting state. The initialGlobalState should be the plain JavaScript representation if the internal state is Immutable.js (using toJS()). [36]
  - **devTools.send(action, newGlobalState)**: This method is called after each action is dispatched and the global state has been updated by the reducers. It sends the action object (or just its type if action is null for lifted state updates) and the new state (again, potentially converted with toJS()) to the DevTools. This allows the DevTools to log the action, display the state diff, and build the history for time-travel. [36]
  - **devTools.subscribe(listener)**: This method registers a listener function that will be called when a message is dispatched *from* the DevTools monitor. These messages can include:
    - 'DISPATCH': Indicates a "time travel" event (e.g., jumping to a previous state, skipping an action, replaying an action). The message.payload

might contain instructions like { type: 'JUMP_TO_STATE', index: targetIndex } or { type: 'TOGGLE_ACTION', id: actionId }. The message.state will contain the stringified state that the application should revert to.
        - The listener needs to parse these messages and update the application's actual global store accordingly (e.g., by replacing the current state or re-computing state based on a series of actions). [36]
    - **devTools.error(message)**: Can be used to send error messages to be displayed in the DevTools monitor.[36]
- Integration Point:
  The logic for DevTools integration would typically reside within the createGlobalStore function or be applied as a store enhancer, similar to applyMiddleware. It needs to wrap the store's dispatch method to call devTools.send() after the actual dispatch and state update. It also needs to handle messages from devTools.subscribe() to implement time-travel functionality.
- Scoped State Consideration:
  Initially, DevTools integration will focus exclusively on the global store. Visualizing and managing a potentially large number of independent, fine-grained scoped states within the Redux DevTools paradigm could become overly complex and might not offer the same level of utility as inspecting the global application state. However, for debugging specific complex components, a mechanism could be explored in the future to allow selected scoped states to send their information to the DevTools, perhaps under a distinct instance ID or as part of a specific "component state" view if the DevTools API supports such custom categorizations.

Leveraging the existing Redux DevTools extension [12] provides immediate access to powerful debugging features like state inspection, action logging, and time-travel debugging, significantly enhancing the developer experience without the need to build proprietary tooling.

**C. State Persistence Layer (Inspired by Killa)**

State persistence allows the application's state (or parts of it) to be saved to a storage medium (like browser localStorage or sessionStorage) and rehydrated when the application is reloaded. This is useful for preserving user preferences, uncompleted forms, or application state across sessions.

- Concept and Killa's Approach:
  Killa.js implements persistence via a persist middleware.3 This middleware can be configured with options such as a name (which becomes the key in storage),

revalidate settings, encrypted flags, and importantly, a storage option that defaults to localStorage but can be customized (e.g., using normalizeStorage(() => sessionStorage) 3).

- Adapter-Based Design for Flexibility:
  To provide maximum flexibility, the persistence layer in this library will adopt an adapter-based design.
  - **Storage Adapter Interface:** A simple interface will be defined for storage adapters, for example:
    ```
    TypeScript
    interface StorageAdapter {
      getItem(key: string): Promise<any> | any;
      setItem(key: string, value: any): Promise<void> | void;
      removeItem(key: string): Promise<void> | void;
    }
    ```
    (Using Promise allows for asynchronous storage mechanisms like IndexedDB.)
  - **Default Adapters:** A default adapter for localStorage will be provided.
  - **Custom Adapters:** Users will be able to supply their own custom storage adapters that conform to this interface. This allows integration with sessionStorage, IndexedDB, server-side storage via API calls, or any other custom persistence mechanism. Killa's normalizeStorage function [3] hints at this kind of adaptability.
- Implementation as Middleware (for Global Store):
  Persistence for the global store can be effectively implemented as a middleware.
  - A persistState middleware would be created. When applied to the global store, it would subscribe to store changes.
  - Upon each state change (or perhaps throttled/debounced for performance), the middleware would:
    1. Select the relevant part of the state to persist (potentially based on a user-provided selector function).
    2. Serialize the selected state (e.g., using JSON.stringify after converting Immutable.js structures to plain JS with toJS()).
    3. Save the serialized state to the configured storage adapter using the specified key.
  - **Rehydration:** During store initialization, the initialState passed to createGlobalStore could be augmented or entirely sourced from the storage adapter. This rehydration logic could be part of the persistState middleware's setup phase or handled explicitly before the store is created.
- Configuration Options:
  The persistState middleware (or a configuration object for it) would accept

options like:
    - key: The string key under which the state is stored (e.g., 'myAppState').
    - adapter: An instance of a storage adapter (e.g., localStorageAdapter).
    - selector: (Optional) A function (state) => persistedStateSlice to specify which part of the global state should be persisted. If omitted, the entire state might be persisted.
    - serializer: (Optional) Custom functions for serialize and deserialize if JSON.stringify/parse and toJS/fromJS are not sufficient.
    - throttleWait: (Optional) Time in milliseconds to throttle save operations.

This adapter-based approach, inspired by Killa's configurable storage [3], ensures that the persistence feature is not tied to a single storage mechanism, making the library more versatile and adaptable to various application needs and environments.

## VI. Synthesizing Strengths: API and Pattern Recommendations

### A. Proposed Core API for the New Library (Initial Version)

This section consolidates the API designs discussed previously into a cohesive public interface for the initial version of the library. The API aims to be intuitive, drawing from established patterns while integrating the unique strengths of the chosen dependencies.

- **Global Store Management:**
    - createGlobalStore({ reducer: RootReducer, initialState?: Immutable.Map<string, any>, enhancer?: StoreEnhancer }): GlobalStore
        - Initializes and returns the global application store.
        - reducer: The root reducer function, typically created using a combineReducers-like utility that handles Immutable.js state slices.
        - initialState: Optional initial state, must be an Immutable.Map.
        - enhancer: Optional function for store enhancement, e.g., applyMiddleware().
    - GlobalStore.getState(): Immutable.Map<string, any>
        - Returns the current Immutable.js state tree of the global store.
    - GlobalStore.dispatch(action: Action): Action
        - Dispatches an action to the global store. The action is processed by reducers to update the state.
    - GlobalStore.subscribe(listener: () => void): () => void
        - Registers a listener function to be called on global state changes. Returns an unsubscribe function.
    - applyMiddleware(...middlewares: Middleware): StoreEnhancer

        - An enhancer to be used with createGlobalStore for applying middleware to the dispatch pipeline.
- **Scoped State Management (VanJS-based):**
    - createScopedState<T>(initialValue: T, options?: { globalStore?: GlobalStore }): ScopedState<T>
        - Creates a reactive scoped state.
        - initialValue: The initial value for the state. Can be a primitive, plain JS object/array, or an Immutable.js structure.
        - options.globalStore: Optional global store instance to enable bridging.
    - ScopedState<T>.val: T (Getter/Setter)
        - Accesses or updates the value of the scoped state, triggering reactivity.
    - ScopedState<T>.oldVal: T (Getter)
        - Provides the previous value of the state.
    - ScopedState<T>.rawVal: T (Getter)
        - Accesses the current value without creating a reactive dependency.
    - deriveScopedState<T>(derivationFn: () => T, options?: { globalStore?: GlobalStore }): DerivedScopedState<T>
        - Creates a derived state whose value is computed by derivationFn based on other states.
        - DerivedScopedState<T>.val: T (Getter) - Accesses the derived value.
- **Bridge API (Methods on ScopedState if globalStore is provided in options):**
    - ScopedState.getGlobal<S>(selectorFn: (globalState: Immutable.Map<string, any>) => S): S
        - Retrieves a slice of the global state using a selector function.
    - ScopedState.dispatchGlobal(action: Action): Action
        - Dispatches an action to the connected global store.
    - ScopedState.subscribeToGlobal<S>(selectorFn: (globalState: Immutable.Map<string, any>) => S, callback: (selectedState: S) => void): () => void
        - Subscribes to changes in a selected slice of the global state. Returns an unsubscribe function.
- **Immutability Utilities (Re-exports from immutable for convenience):**
    - Immutable: The re-exported immutable library itself (e.g., Immutable.Map, Immutable.List).
    - fromJS<T>(jsValue: any): T (where T is an Immutable.js type)
        - Re-exports Immutable.fromJS() for converting plain JS to immutable structures.
    - isImmutable(maybeImmutable: any): boolean

- Re-exports Immutable.isImmutable().

The following table maps these proposed API methods to their primary source of inspiration and the key dependencies they will utilize, providing a clear rationale for each choice. This transparency is vital for understanding the initial construction of the library by "piece mealing" functionality from existing, robust solutions.

**Table 2: Key API Mapping for Initial Implementation**

| Proposed Library API Method/Concept | Primary Source Library & API | Key Dependency Used | Brief Description & Rationale for Choice |
|---|---|---|---|
| createGlobalStore(reducer, initialState, enhancer) | Redux: createStore [6] | redux, immutable | Provides a robust, well-understood global state foundation. enhancer supports middleware. State managed with immutable. |
| globalStore.dispatch(action) | Redux: store.dispatch [6] | redux | Standard Redux pattern for initiating state changes via actions, crucial for traceability and middleware. |
| globalStore.getState() | Redux: store.getState [6] | redux, immutable | Standard Redux method to access the current state. Returns an Immutable.js structure. |
| globalStore.subscribe(listener) | Redux: store.subscribe [6] | redux | Standard Redux pattern for reacting to state changes. |
| applyMiddleware(...middlewares) | Redux: applyMiddleware [8] | redux | Standard Redux enhancer for extending store capabilities with |

| | | | |
|---|---|---|---|
| | | | custom logic (logging, async). |
| createScopedState(initialValue,?options) | VanJS: van.state [4] | vanjs-core | Offers reactive, lightweight state for UI components with direct DOM binding. Options allow bridging to global store. |
| scopedState.val (getter/setter) | VanJS: state.val [4] | vanjs-core | Core VanJS mechanism for accessing and updating reactive state. |
| deriveScopedState(derivationFn,?options) | VanJS: van.derive [5] | vanjs-core | VanJS mechanism for creating computed states that react to changes in their dependencies. |
| scopedState.getGlobal(selectorFn) | Inspired by React-Redux useSelector / mapStateToProps [25] | immutable (for selector input) | Provides controlled read-only access from scoped state to the global Immutable.js state. |
| scopedState.dispatchGlobal(action) | Inspired by React-Redux useDispatch / mapDispatchToProps [25] | redux (via global store) | Allows scoped components to initiate changes in the global store following Redux patterns. |
| scopedState.subscribeToGlobal(selectorFn, callback) | Custom, inspired by store subscriptions and selectors | redux, immutable | Enables scoped states to react efficiently to specific changes in the global store. |
| Immutable | Immutable.js: | immutable | Provides direct access to |

| | | | |
|---|---|---|---|
| (re-export) | Immutable object [2] | | Immutable.js data structures (Map, List) for users. |
| fromJS() (re-export) | Immutable.js: fromJS() [2] | immutable | Essential utility for converting plain JavaScript data into immutable structures for the store. |

**B. Key Architectural Patterns for Integrating Dependencies**

To ensure the library is maintainable and can evolve towards its dependency-free goal, specific architectural patterns will be employed:

- **Facade Pattern:** The public API of this library will act as a facade over the underlying dependencies. For example, users will call myLibrary.createGlobalStore() rather than directly invoking redux.createStore(). This abstraction is paramount. It decouples the library's users from the internal implementation details and the specific dependencies being used in any given version. If Redux is later replaced by a custom solution, as long as the facade (myLibrary.createGlobalStore()) maintains its contract (signature and behavior), user applications will not require modification. This is a cornerstone for achieving the long-term vision of a dependency-free library with minimal disruption to its consumers.
- **Adapter Pattern:** For functionalities that might have multiple underlying implementations or interact with external systems, the Adapter pattern will be used. State persistence is a prime example: by defining a storage adapter interface, the library can support localStorage, sessionStorage, IndexedDB, or even custom server-side storage solutions, simply by providing a conforming adapter. This pattern promotes flexibility and extensibility.
- **Modular Design:** The integration logic for each core dependency (Redux, VanJS, Immutable.js) will be encapsulated within distinct modules of the new library (e.g., globalStore.js for Redux/Immutable.js, scopedState.js for VanJS). This separation of concerns simplifies development, testing, and future refactoring. When a dependency is eventually replaced, the changes are localized to its corresponding module.

**C. Leveraging Learnings from the Dependency-Free Research** [1]

The prior research conducted for a dependency-free version of this library [1] remains highly relevant, even when initially building with dependencies.

- **API Design with Future in Mind:** The public API exposed by the facade should be designed from the perspective of the ideal, dependency-free library. Even if a function currently wraps a method from an external library, its signature and behavior should align with the long-term vision. This foresight will minimize breaking changes when dependencies are eventually removed or replaced.
- **Core Logic Separation and Incremental Implementation:** The dependency-free research may have identified fundamental, reusable logic (e.g., a custom publish/subscribe mechanism, a basic reactive primitive). If feasible, these small, core pieces could be implemented now and either used by the modules that wrap external dependencies or sit alongside them. For instance, the global store's subscribe method could internally use a custom-built event emitter (from the dependency-free design) even if the core state processing still relies on Redux. This allows for an incremental path towards being dependency-free, testing custom components within a functioning system before undertaking larger dependency replacements.
- **Performance Considerations:** Insights into performance bottlenecks or optimizations from the dependency-free research [1] should inform the current design. If a chosen dependency introduces a known performance issue for a common use case that the dependency-free approach would handle more efficiently, this should be noted and prioritized for future refactoring when moving away from that dependency.

By strategically applying these architectural patterns and continuously referencing the goals of the dependency-free research, the library can be developed rapidly using existing tools while simultaneously paving a clear and manageable path towards its ultimate self-reliant form.

## VII. Conclusion and Strategic Next Steps

### A. Summary of Proposed Architecture and Benefits

The proposed architecture outlines a hybrid vanilla JavaScript state management library that strategically leverages the strengths of established dependencies—Redux, VanJS, and Immutable.js—to achieve rapid initial development and a robust feature set. It features a dual-store model: a Redux-inspired global store ensuring predictable, application-wide state management with mandatory immutability via Immutable.js; and a VanJS-based scoped state mechanism for lightweight, reactive UI component state with direct DOM binding. A well-defined bridge facilitates seamless, controlled interaction between these two scopes.

The primary benefits of this approach include:

- **Accelerated Development:** Reusing proven functionalities from dependencies significantly shortens the time to a viable first version.
- **Rich Initial Feature Set:** Users gain access to sophisticated capabilities like predictable global state, reactive local state, and enforced immutability from day one.
- **Clear Evolutionary Path:** The architecture is designed with future enhancements (middleware, DevTools, persistence) and the long-term goal of becoming dependency-free in mind, primarily through abstraction layers like the Facade pattern.

**B. Phased Development Recommendations**

A phased development approach is recommended to manage complexity and ensure stability:

- **Phase 1 (Core Functionality):**
  - Implement the createGlobalStore module, integrating redux and Immutable.js. This includes dispatch, getState, subscribe, and the action/reducer pattern with Immutable.js state slices.
  - Implement the createScopedState module, leveraging vanjs-core for reactive local states (.val, deriveScopedState).
  - Develop the bridge mechanism allowing scoped states to read from and dispatch to the global store.
  - Establish a basic API as outlined in Table 2, focusing on core interactions.
  - Thorough unit and integration testing for these core components.
- **Phase 2 (Enhancements & Developer Experience):**
  - Implement the applyMiddleware system for the global store, including a basic logging middleware and a thunk-like middleware for simple asynchronous operations.
  - Integrate the global store with Redux DevTools via the window.__REDUX_DEVTOOLS_EXTENSION__ API.
  - Develop the initial state persistence layer for the global store, with a default localStorage adapter and the ability to use custom adapters.
- **Phase 3 (Expansion, Refinement, and Initial Dependency Reduction):**
  - Develop a comprehensive build system (e.g., using esbuild or Rollup) to generate various distribution packages (ESM, UMD, CJS) and type definitions (if TypeScript is used for development).
  - Begin exploring the replacement of smaller, well-contained dependency functionalities with custom code, guided by the dependency-free research.[1]

For example, if a very specific utility from a larger library is used, it might be a candidate for an early custom implementation.
    - Refine APIs based on early usage feedback and expand utility functions.

**C. Addressing Long-Term Dependency-Free Goal**

The transition to a dependency-free library is a strategic long-term objective. The architectural choices made in the initial phases are crucial for enabling this:

- **Facade Pattern:** As emphasized, the public API must act as an abstraction layer. This allows internal dependencies to be swapped out without breaking changes for library users.
- **Modular Design:** Encapsulating dependency-specific logic within dedicated modules localizes the impact of future replacements.
- **Prioritized Replacement:** A roadmap for replacing dependencies should be established. This might prioritize:
    1. Dependencies with the most significant performance impact or licensing concerns.
    2. Dependencies whose functionality is relatively straightforward to replicate with custom code.
    3. Dependencies that are less central to the core value proposition once the library has established its own patterns. The dependency-free research [1] will be instrumental in guiding these custom implementations.

**D. Final Considerations**

- **Testing:** Rigorous and comprehensive testing (unit, integration, and end-to-end where applicable) is paramount throughout all development phases.
- **Documentation:** Clear, concise, and comprehensive documentation, including API references, usage examples, and architectural explanations, will be vital for adoption and community engagement.
- **Community and Contribution:** If the library is intended for open-source release, establishing clear contribution guidelines, a code of conduct, and channels for community discussion early on will foster healthy growth.
- **Minimal Viable Product (MVP):** The initial focus should be on delivering a stable and functional MVP that validates the core concepts of the global store, scoped state, and their interoperability. Advanced features can be layered on this solid foundation.

By following this strategic approach, the project can successfully deliver a powerful and versatile state management library that meets immediate needs through intelligent dependency usage, while being well-positioned for future evolution into a

fully independent solution.

**Works cited**


1. accessed December 31, 1969, https://docs.google.com/document/d/1Bw4r1G0DPrJEbZlsaDOf9Aj3r68mEPxYslHcEaXHLiw/edit?tab=t.0
2. Immutable.js, accessed June 4, 2025, https://immutable-js.com/
3. JesuHrz/killa: Killa is a small and lightweight state ... - GitHub, accessed June 4, 2025, https://github.com/JesuHrz/killa
4. VanJS (Legacy) - Tutorial and API Reference, accessed June 4, 2025, https://legacy.vanjs.org/tutorial
5. Tutorial and API Reference - VanJS, accessed June 4, 2025, https://vanjs.org/tutorial
6. What are the 3 core concepts of React Redux - GeeksforGeeks, accessed June 4, 2025, https://www.geeksforgeeks.org/what-are-the-3-core-concepts-of-react-redux/
7. Learn Redux: Core Concepts in Redux Cheatsheet | Codecademy, accessed June 4, 2025, https://www.codecademy.com/learn/learn-redux/modules/core-concepts-in-redux/cheatsheet
8. Store | Redux, accessed June 4, 2025, https://redux.js.org/api/store
9. Immutable Data | Redux, accessed June 4, 2025, https://redux.js.org/faq/immutable-data
10. Middleware - Redux Documents, accessed June 4, 2025, https://redux.ruanyifeng.com/advanced/Middleware.html
11. Redux Fundamentals, Part 4: Store, accessed June 4, 2025, https://redux.js.org/tutorials/fundamentals/part-4-store
12. Redux DevTools Overview - Tutorialspoint, accessed June 4, 2025, https://www.tutorialspoint.com/redux/redux_devtools.htm
13. Store - Redux Documents, accessed June 4, 2025, https://redux.ruanyifeng.com/api/Store.html
14. combineReducers | Redux, accessed June 4, 2025, https://redux.js.org/api/combinereducers
15. `combineReducers(reducers)` - Redux Documents, accessed June 4, 2025, https://redux.ruanyifeng.com/api/combineReducers.html
16. Redux store as an immutable.js map - Stack Overflow, accessed June 4, 2025, https://stackoverflow.com/questions/42916286/redux-store-as-an-immutable-js-map
17. Push or Add with the Immutability Operation in Redux and Immutable.js - Pluralsight, accessed June 4, 2025, https://www.pluralsight.com/resources/blog/guides/pushadd-with-immutable-in-reduximmutablejs
18. How to handle Immutable State in Redux Reducers? - GeeksforGeeks, accessed June 4, 2025,

https://www.geeksforgeeks.org/how-to-handle-immutable-state-in-redux-reducers/
19. Getting started with immutable.js in your React apps | Jeff Lau's website, accessed June 4, 2025, https://jefflau.net/2016-07-30-getting-started-with-immutable-js-in-your-react-apps/2016-07-30-getting-started-with-immutable-js-in-your-react-apps/
20. How to use Immutable.js with redux? - javascript - Stack Overflow, accessed June 4, 2025, https://stackoverflow.com/questions/31892767/how-to-use-immutable-js-with-redux
21. accessed December 31, 1969, https://github.com/redux-immutable/redux-immutable
22. Hello and welcome | Van Element - Docs, accessed June 4, 2025, https://van-element.pages.dev/intro/get-started
23. Tutorial | Van Element - Docs, accessed June 4, 2025, https://van-element.pages.dev/intro/tutorial
24. Using custom elements - Web APIs | MDN, accessed June 4, 2025, https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_custom_elements
25. How to Implement Global State Management with Redux - PixelFreeStudio Blog, accessed June 4, 2025, https://blog.pixelfreestudio.com/how-to-implement-global-state-management-with-redux/
26. Using Redux to Make a Global State - DEV Community, accessed June 4, 2025, https://dev.to/miriamfark/using-redux-to-make-a-global-state-38c8
27. Managing complex state in vanilla JavaScript - Latenode community, accessed June 4, 2025, https://community.latenode.com/t/managing-complex-state-in-vanilla-javascript/13342
28. How do I find out which DOM element has the focus? - Stack Overflow, accessed June 4, 2025, https://stackoverflow.com/questions/497094/how-do-i-find-out-which-dom-element-has-the-focus
29. HTMLElement: focus() method - Web APIs | MDN, accessed June 4, 2025, https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/focus
30. When to use state management libraries? : r/reactjs - Reddit, accessed June 4, 2025, https://www.reddit.com/r/reactjs/comments/1anza21/when_to_use_state_management_libraries/
31. This Library Makes State Management So Much Easier - YouTube, accessed June 4, 2025, https://m.youtube.com/watch?v=s0h34OkEVUE&pp=ygUPI3N0YXRlX2luX3JlYWN0
32. Handle Events | Communicate with Events | Lightning Web Components Developer Guide, accessed June 4, 2025,

https://developer.salesforce.com/docs/platform/lwc/guide/events-handling.html
33. devagrawal09/solid-events: Declarative event composition and state derivation primitives for Solidjs - GitHub, accessed June 4, 2025, https://github.com/devagrawal09/solid-events
34. Mastering Efficient DOM Manipulation with Virtual DOM and Refs - GeeksforGeeks, accessed June 4, 2025, https://www.geeksforgeeks.org/mastering-efficient-dom-manipulation-with-virtual-dom-and-refs/
35. applyMiddleware | Redux, accessed June 4, 2025, https://redux.js.org/api/applymiddleware
36. redux-devtools/extension/docs/API/Methods.md at main · reduxjs ..., accessed June 4, 2025, https://github.com/reduxjs/redux-devtools/blob/main/extension/docs/API/Methods.md
37. How to Debug a React Context API App - Bruno Sabot, accessed June 4, 2025, https://brunosabot.dev/posts/2020/how-to-debug-a-react-context-api-app/
38. accessed December 31, 1969, https://github.com/reduxjs/redux-devtools/blob/main/extension/docs/API/Store.md
39. Best Redux Developer Tools for Increasing Productivity - Turing, accessed June 4, 2025, https://www.turing.com/kb/redux-developer-tools-for-redux-programmers