State Management Standard X (draft 1)

Keith Scroggs

15 August 2020

Contents

Code Examples

1 Goals of this Document

This document was written to provide a standard for the usage and development of client-side state management that follows three central principles:

1. Efficiency: State management code written following this standard and its best practices will be efficient with its usage of available computer resources.
2. Predictability: State management code written following this standard will be predictable and therefore easier to troubleshoot.
3. Ease of Understanding: Code written following this standard will be understood with relative ease by other programmers that read and understand this standard.

These principles were established as both a grading system for quality of a state management system, and an enumeration of the three biggest challenges when writing state management code. Following this standard is recommended for anyone working with unopinionated state management tools, such as Redux, Vuex, and Svelte, in order to write more consistent code that follows the principles stated above.

2 Definitions

2.1 Interpretation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. The JSON format is to be be interpreted according to living IETF standard 90.

2.2 Terminology

In this document, these terms will be used repeatedly to refer to specific concepts. These concepts are agnostic to language, framework, and implementation, although examples will be provided using JavaScript and Svelte.

A "valid data structure" refers to any piece of data that meets the following requirements:

1. Contains a key-value system for accessing child members.
2. Only contains keys that are UTF-8 encoded strings.
3. Can be exported as a JSON encoded string with the use of no more than two transformations.

Any time a valid data structure contains an ID as a child member, the key for that member MUST be named `id`.

"State" refers to the relevant data needed to ensure correct and accurate operation of any program. **"Resource"** refers to any valid data structure that can be accessed by a unique identifier (ID). "Store" refers to a valid data structure containing similar resources as child members. A state is composed of as many stores as needed, which are composed of many resources.

A "parent data source" refers to any source of data that a store MUST be able to both read and write to. Examples of this include localstorage, IndexedDB, CRUD capable APIs, and JSON/CSV files. If a parent data source contains encrypted data, it SHOULD provide clients with checksums of the encrypted data, so that the client may avoid wasteful decryption operations if the data has not changed from a cached version.

A "volatile store" refers to any store that operates independently of any other data source, and consequently results in the loss of any data contained in the store when it is removed from memory. A "nonvolatile store" refers to any store that operates dependently on another data source. A nonvolatile store MUST use the commit method outlined in subsection 4.4 to save data to its parent data source. A nonvolatile store's data source SHOULD be nonvolatile in itself, meaning that it is not entirely stored in memory.

"Passing" refers to when a method returns as soon as available, without performing any interactions or operations on any data or throwing/returning any form of error.

3 Resource & Store Structure

3.1 Resource Structure

Each resource MUST be a valid data structure as defined in subsection 2.2. A resource MAY include child members that cannot be exported as a string without the loss of information. An example of this is that a resource may include a CryptoKey object, which will be exported as `[object Object]`. Each resource MUST have a unique resource identifier (ID). This ID MAY be included as a child member of the resource. This ID MUST be included as a key used to access the resource from its parent store.

Listing 1: Valid Structures

```
// Allowed
'123': { // Where '123' is a unique ID
    title: String,
    description: String,
    key: CryptoKey
}
'123': {
    title: String,
    description: String,
    key: CryptoKey,
    id: '123'
}

// Disallowed
true: {
}
'123': {
    wrongMember: '123'
}
'123': []
```

From henceforth, a store will be referred to as three components: data, methods, and reactivity. A store's data refers to the underlying state that the store is used to access and subscribe to. A store's methods refer to callable code that is used to modify a store's data. A store's reactivity refer to any internal components of a store that are used to notify subscribers when a method changes data.

Listing 2: Data, Reactivity, and Methods

```
import { writable } from 'svelte/store'

function create() {
    const users = {} // Data
    const { subscribe, update } = writable(users)

    return {
        subscribe, // Reactivity
        // Methods
    }
}

```

```
// Since there are no derived stores from the users stores,
// allowing for named imports is not required, but it is still best practice
export const users = create()

export default users
```

3.2 Store Data

A store's data MUST be a valid data structure. Each key MUST be an ID that corresponds to a resource, each key MUST be accurate to the resource it is used to access. A store MUST NOT contain any additional keys.

3.3 Store Methods

A store MUST contain the methods defined in subsection 4.1. These methods MUST behave as defined in subsection 4.2. A store MAY contain a method (private or public) that is called to subscribe a point of access to changes in the store's data. Once a client is subscribed, the client MUST be notified when any method that manipulates relevant data is called. A compiler MAY surgically transform subscriptions so that clients are only notified of the information that they access during their memory scope. A runtime MUST NOT emulate this behavior.

3.4 Store Reactivity

A store MAY contain reactivity, a mostly invisible component dedicated to allowing points of access (clients) to "subscribe" to the store, and notifying these clients when a change is made. A store's reactivity MUST notify any subscribers when the store's data is updated using a method. A store's reactivity MAY notify any subscribers when the store's data is updated without the usage of a method. Clients SHOULD NOT rely that a store will notify them when data is updated without the use of a method.

4 Methods

4.1 Required Methods

A store MUST contain methods `create(data): { id }, update(id, data): void, delete(id): void`. If a store is classified as nonvolatile, it MUST contain methods `init(): void, commit(id): void`. These methods MUST behave as outlined in the following subsections. Any other methods MAY be created to behave as desired by the programmer, as long as this behavior includes and does not conflict with the behavior defined in the following subsection.

4.2 Method Behavior

Any method of a store, whether the method is required by this standard or custom, must adhere to and not conflict with the behavior defined in this section. A method MUST NOT be assumed to succeed. A method MUST either throw or return any relevant errors.

In languages that support async/await syntax, any method of a store that accesses async functionality MUST be declared as async. This is to ensure that any rejected promises are properly passed upstream so that they may be handled.

If a method accepts an ID as a parameter, the method SHOULD verify if the ID actually refers to a resource belonging to the same store as the method being called before performing any interactions/operations. A method SHOULD throw/return an error of an ID is passed that does not match a child member of the store from which the method is being called. If no ID is passed, the method MAY choose to throw an error, otherwise failing silently.

If a method accepts a valid data structure as a parameter in a non-typesafe language, the method SHOULD validate the structure according to the three requirements outlined in subsection 2.2. A method MAY choose to perform custom validation according to a custom schema. If any validation fails, the method MUST throw/return an error.

If a method performs interactions with a parent data source, the method MUST either inform the caller of any information passed down from the parent data source, or update the relevant resource to contain this information. If the parent data source returns any data calculated based on a request made during the method's calling (i.e checksums), the method MUST either update the value of the resource specified to contain that data, or return a valid data structure containing the data returned by the parent data source. A method MUST NOT ignore ANY data returned by a parent data source.

A method MUST throw/return an error if interacting with a parent data source fails in any way. If this error is passed down from the parent data source, the method MUST NOT intercept or alter this error in any way.

4.3 Initialization

If a store is nonvolatile, it MUST contain a method to initialize itself from its parent data source(s). This method MUST be named `init`.

This method MUST accept no arguments and return no value (except for an error if applicable).

If the store is nonvolatile, it MAY omit the initialization method. If a store is nonvolatile and chooses not to omit the initialization method, the method MUST pass.

Listing 3: Initialization Method

```
async init() {
    const res = await fetch('https://api.example.com/resources', {
        method: 'GET'
    })
    const body = await res.json()

    for (const resource of body) {
        update((store) => {
            store[resource.id] = resource
            return store
        })
    }
}
```

4.4 Committing to a Parent Data Source

If a store is nonvolatile, it MUST contain a method to commit an updated resource to its parent data source. This method MUST be named `commit`.

This method MUST accept one argument, this argument being the ID of the resource being committed.

This method MUST ensure that any and all parent data sources are updated to contain the same version of the resource as the store does at the time of calling.

If a store is volatile, it MAY omit the commit method. If a store is nonvolatile and chooses not to omit the commit method, the method MUST pass.

Listing 4: Commit Method

```
async commit(id) {
    const resources = get(resourceStore)
    if (!resources[id]) {
        throw new Error('Invalid ID')
    }
    const res = await fetch(`https://api.example.com/${resource.id}`, {
        method: 'PATCH',
        body: JSON.stringify(resources[id])
    })
    const body = await res.json()
    if (res.status !== 200) {
        throw new Error(body.message || 'Unknown error')
    }
    update((store) => {
        store[id].checksum = body.meta.checksum
    })
    // OR
    return {
        checksum: body.meta.checksum
    }
}
```

4.5 Creating a New Resource

A store MUST contain a method to create a new resource. This method MUST be named `create`.

This method MUST accept one argument, this argument being a valid data structure. This data structure SHOULD contain all information available about the resource being created. This method MAY choose to validate the data structure before performing any operations, and throw/return an error if validation is unsuccessful.

This method MUST ensure that the creation of the resource is propagated upstream to any parent data sources before returning. This method MUST return a valid data structure containing at least a member named `id`, containing the resource's created ID. Regarding additional members, this method must adhere to subsection 4.2, paragraphs 4-5.

Listing 5: Create Method

```
async create(data) {
    const validated = customValidator(data)
    if (!validated) {
        throw new Error('Bad data supplied')
    }

    const res = await fetch('https://api.example.com/resources', {
        method: 'POST',
        body: JSON.stringify(data)
    }
    const body = await res.json()
    if (!res.status === 201) {
        throw new Error(body.message || 'Unknown error')
    }

    update((store) => {
        store[body.id] = { ...data, id: body.id, checksum: body.checksum }
    })

    return {
        id: body.id,
```

```
22              checksum: body.checksum
23          }
24      }
```

4.6 Updating an Existing Resource

A store MUST contain a method to update an existing resource. This method MUST be named `update`.

This method MUST accept two arguments, the first argument being the ID of the resource being updated, and the second argument being a valid data structure containing the data to update the resource with.

This method MUST treat omitted child members as containing their last known value. This method MUST NOT treat omitted child members as containing empty, null, or undefined values. This method MUST update ONLY the resource specified by ID in the first parameter, using the data specified in the second parameter.

This method MAY choose to immediately commit the updated resource after the update is complete. If this behavior is desired, the commit method MUST be called using the same ID that was passed to this method. This method MUST NOT emulate the commit method within its own code. Whether or not a store should choose to commit each update SHOULD be decided based on how frequently it updated, whether there are multiple triggers for updates and commits (i.e triggering an update on input, and a commit on blur), and what is deemed to strike an appropriate balance between simplicity of implementation and performance for end users.

Listing 6: Update Method

```
update(id, data) {
    // Looks weird, but namespacing is a beautiful thing
    update((store) => {
        store[id] = { ...store[id], ...data } // Retain any properties not updated in the
            data passed
    })
    // Not required
    await this.commit(id)
}
```

4.7 Deleting an Existing Resource

A store MUST contain a method to delete an existing resource. This method MUST be named `delete`.

This method MUST accept one argument, this argument being the ID of the resource to be deleted.

This method MUST ensure that the deletion of the resource is propagated upstream to any parent data sources before returning. If any step of the deletion fails, this method MAY choose to throw an error or fail silently. Whether this method should fail silently or loudly SHOULD be decided based on whether a handler is available for the event of an error.

Listing 7: Delete Method

```
async delete(id) {
    update((store) => {
        delete store[id]
        return store
    })

    const res = await fetch('https://api.example.com/resources' + id, {
        method: 'DELETE'
    })
    if (res.status !== 204) {
        // There will only be a body if there's an error message to give in this case
        const body = await res.json()
        throw new Error(body.message || 'Deletion from API failed')
    }
}
```

5 Derivation & Exports

Note: this section is only applicable to state management systems which implement derivation at the *store level*.

5.1 Allowance

A store MAY implement derivations/computations of itself. Unlike regular stores, derived stores are NOT REQUIRED to be a valid data structure. In fact, a primary use case of derived stores is to transform a store from a valid data structure to another format (such as an array). If a derived store is not in the format of a valid data structure, it MUST adhere to the guidelines of subsection 5.2. A derived store SHOULD be declared in the same file in which the parent store is declared.

5.2 Alternate Data Structures

A common use case for derived stores is converting a store into a different data structure for purposes of iteration, display to end users, debugging, etc. If a derived store is not a valid data structure, and contains multiple resources, it must obey the following guidelines:

1. Each resource MUST be kept separate by means which the language parser used will understand (no custom separators). A resource's properties MAY be kept separate, but this is not required.
2. Each resource's ID MUST be preserved as a child member named `id` (post standard parsing if applicable).
3. When a resource is retrieved by means of iteration, its ID MUST be retrievable as specified in guideline #2, and the ID retrieved MUST be usable to access the resource in the parent store.

Listing 8: Derived Store as Array

```
// Allowed
[
    {
        id: '1234',
        title: 'atitle',
        description: 'adescription'
    }
]
// JavaScript can parse JSON without any special/unstandard logic, so this is still valid separation.
'[{"id":"1234","title":"atitle","description":"adescription"}]'
[
    id: '1234',
    titleAndDescription: 'atitle-adescription'
]



// Disallowed
// Violation of guideline #1
['1234,5678,9101112', 'onetitle,twotitle,threetitle', 'adescription,anotherdescription']
[
    ['1234', 'atitle', 'description'] // Violation of guideline #2
]
[
    {
        // Violation of guideline #3
        id: 'm', // ID is encoded differently from the parent store
        title: 'atitle',
        description: 'adescription'
    }
]
```

5.3 Exports

If a derived store is declared in the same file as its parent store, this file MUST utilize named exports for each store (including the parent store). This file MAY still specify a default export, this SHOULD be the parent store for semantic purposes, but it MAY be any store declared and exported in the file. If a file declares a default export, it MUST be declared as the last export in the file, below all other exports.

5.3.1 Create Functions

A create function is defined as any function that creates a store, and will not return an already existing store if called again. Any function that meets this definition MUST NOT be called more than once during the lifetime of the program. If a file contains named exports and a default export, the default export MUST be set to a variable already exported as a named export, it MUST NOT call a create function again.

Listing 9: Usage of Create Functions and Named Exports

```
// Allowed
export const resources = create()
export const derivedResources = createDerived()

export const resources = create()
export const derivedResources = createDerived()
export default resources

// Allowed, but semantically flawed
export const resources = create()
export const derivedResources = createDerived()
export default derivedResources

// Disallowed
export const resources = create()
export const derivedResources = createDerived()
export default create()
```

6 Application of this Document

Although this document tries to be agnostic in its terminology, this standard specifically written for client-side state management. Trying to implement this standard for any other forms of state management or data handling, such as APIs, databases, or cloud infrastructure, will likely result in frustration, and is not advised.

6.1 For Developers

Programs implementing this standard are highly encouraged to specify their compliance, and ideally link back to the version of this standard which their compliance has been verified with. This will help other developers to gain a better understanding of the program, and refer more people to this standard. The source code for this standard is open source and written in LaTeX, anyone that would like to submit changes, issues, or suggestions is encouraged to do so at https://github.com/very-amused/SMSX.