

# State Management Standard X (draft 1)

Keith Scroggs

15 August 2020

## Contents

<b>1</b>	<b>Goals of this Document</b>	<b>3</b>
<b>2</b>	<b>Definitions</b>	<b>3</b>
2.1	Interpretation . . . . .	3
2.2	Terminology . . . . .	3
<b>3</b>	<b>Resource &amp; Store Structure</b>	<b>4</b>
3.1	Resource Structure . . . . .	4
3.2	Store Data . . . . .	5
3.3	Store Methods . . . . .	5
3.4	Store Reactivity . . . . .	5
<b>4</b>	<b>Methods</b>	<b>6</b>
4.1	Method Behavior . . . . .	6
4.2	Required Methods . . . . .	6
4.3	Creating a New Resource . . . . .	7
4.4	Updating an Existing Resource . . . . .	7
4.5	Deleting an Existing Resource . . . . .	8
4.6	Initialization . . . . .	9
4.7	Committing to a Parent Data Source . . . . .	9
<b>5</b>	<b>Derivation &amp; Exports</b>	<b>10</b>
5.1	Allowance . . . . .	10
5.2	Alternate Data Structures . . . . .	10
5.3	Exports . . . . .	11
5.3.1	Create Functions . . . . .	11

<b>6</b>	<b>Caching of Encrypted Resources</b>	<b>11</b>
<b>7</b>	<b>Application of this Document</b>	<b>12</b>
7.1	For Developers . . . . .	12

## Code Examples

1	Valid Resource Structure . . . . .	4
2	Data, Reactivity, and Methods . . . . .	5
3	Create Method . . . . .	7
4	Update Method . . . . .	8
5	Delete Method . . . . .	8
6	Initialization Method . . . . .	9
7	Commit Method . . . . .	9
8	Derived Store as Array . . . . .	10
9	Usage of Create Functions and Named Exports . . . . .	11

# 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 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 not specific to language, framework, and implementation, although examples will be provided using JavaScript and Svelte.

**Map** refers to a data structure 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.

Language specific types that meet this definition include JavaScript's *Object*, Go's *Struct* and *Map*, and Python's *Dict*. Child members of maps will be referred to as *properties* for the purpose of this document.

**ID** refers to a unique *string* that is used to identify a resource.

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

**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.

**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. **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 map. A resource **MAY** include properties that cannot be exported as JSON 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 property of the resource. This ID **MAY** be generated within the store itself or by a parent data source. This ID **SHOULD NOT** be enumerated, a random number generator **SHOULD** be used to obtain this ID, and its uniqueness within the resource's parent store **MUST** be guaranteed, regardless of how/where it was generated. This ID **MUST** be included as a key used to access the resource from its parent store.

Resources **SHOULD NOT** contain maps with subproperties. If a parent data source returns a map with multiple depths, the store **SHOULD** flatten all properties needed into a single map before storage or returning any information.

Listing 1: Valid Resource Structure

```
1 // Allowed
2 '123': { // Where '123' is a unique ID
3     title: String,
4     description: String,
5     key: CryptoKey
6 }
7 '123': {
8     title: String,
9     description: String,
10    key: CryptoKey,
11    id: '123'
12 }
13
14 // Allowed, but not recommended
15 '123': {
16     title: String,
17     description: String,
18     meta: {
19         key: CryptoKey
20     }
21 }
22
23 // Disallowed
24 true: {
25 }
26 '123': {
27     wrongMember: '123'
28 }
29 '123': []
```

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

```
1 import { writable } from 'svelte/store'
2
3 function create() {
4   const users = {} // Data
5   const { subscribe, update } = writable(users)
6
7   return {
8     subscribe, // Reactivity
9     // Methods
10  }
11 }
12
13 // Since there are no derived stores from the users stores,
14 // allowing for named imports is not required, but it is still best practice
15 export const users = create()
16
17 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 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 (calls any function that is declared as `async` or returns a promise) **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 **MUST** 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 **MUST** throw/return an error if an ID is passed that does not match a resource belonging to the store from which the method is being called. If no ID is passed, the method **MUST** throw an error.

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

Methods **MUST NOT** specify default values for any arguments, as this would result in less specificity required for a successful call to the method, and therefore less predictable behavior.

If a method performs interactions with a parent data source, the method **MUST** either return all data passed down from the parent data source, or update the relevant resource to contain this information. These two options **MAY** be combined (some information passed down from a parent source may be added to the resource, and some may be returned, as long as no data is ignored). 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. A method **MUST** account for the case of being unable to parse an error from the parent data source, providing a fallback error to throw/return if this does occur. This error **SHOULD** contain a level of verbosity that describes what stage of the method failed at minimum.

### 4.2 Required Methods

A store **MUST** contain the following methods:

- `create(data): { id }`
- `update(id, data): void`
- `delete(id): void`

If a store is classified as nonvolatile, it **MUST** contain the following methods in addition to the methods required above:

- `init(): void`
- `commit(id): { calculatedData } | void`

The names of the methods shown above are **REQUIRED** and **RESERVED**. The names of arguments shown above are **NOT REQUIRED**.

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.

### 4.3 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 successfully performed upstream by any parent data source(s) before performing the creation within the store. This method **MUST** return a map 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 3: Create Method

```
1  async create(data) {
2      const validated = customValidator(data)
3      if (!validated) {
4          throw new Error('Bad data supplied')
5      }
6
7      const res = await fetch('https://api.example.com/resources', {
8          method: 'POST',
9          body: JSON.stringify(data)
10     })
11     const body = await res.json()
12     if (!res.status === 201) {
13         throw new Error(body.message || 'Failed to create resource (status ${res.status})')
14     }
15
16     update((store) => {
17         store[body.id] = { ...data, id: body.id, checksum: body.checksum }
18     })
19
20     return {
21         id: body.id,
22         checksum: body.checksum
23     }
24 }
```

### 4.4 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. If the commit method is declared as async or returns a promise, the update method **MUST** be declared as async, and the commit method **MUST** be awaited. 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 4: Update Method

```
1 update(id, data) {
2   // Looks weird, but namespacing is a beautiful thing
3   update((store) => {
4     store[id] = { ...store[id], ...data } // Retain any properties not updated in the
        data passed
5   })
6   await this.commit(id) // Not required
7 }
```

## 4.5 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 performed upstream by any parent data source(s) before deleting it from the local store. If any step of the deletion fails, this method **MAY** choose to throw an error or fail silently. If the upstream deletion of the resource fails, this method **MUST** fail silently or loudly without making any changes to the store. 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 5: Delete Method

```
1 async delete(id) {
2   const res = await fetch('https://api.example.com/resources' + id, {
3     method: 'DELETE'
4   })
5   if (res.status !== 204) {
6     // There will only be a body if there's an error message to give in this case
7     const body = await res.json()
8     throw new Error(body.message || 'Deletion from API failed')
9     // This method can choose to fail silently
10    return
11  }
12
13  update((store) => {
14    delete store[id]
15    return store
16  })
17 }
```



## 4.6 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 data returned from a parent data source is encrypted, this method is where the store **SHOULD** use checksums to compare the version of encrypted data retrieved to a decrypted cached version. More detail regarding caching is available in section 6.

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 6: Initialization Method

```
1 async init() {
2   const res = await fetch('https://api.example.com/resources', {
3     method: 'GET'
4   })
5   const body = await res.json()
6
7   for (const resource of body) {
8     update((store) => {
9       store[resource.id] = resource
10      return store
11    })
12  }
13 }
```

## 4.7 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 updated version of the resource before making any changes to the store.

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 7: Commit Method

```
1 async commit(id) {
2   const resources = get(resourceStore)
3   if (!resources[id]) {
4     throw new Error('Invalid ID')
5   }
6   const res = await fetch('https://api.example.com/${resource.id}', {
7     method: 'PATCH',
8     body: JSON.stringify(resources[id])
9   })
10  const body = await res.json()
11  if (res.status !== 200) {
12    throw new Error(body.message || 'Failed to commit changes to API (status ${res.status})')
13  }
14  update((store) => {
15    store[id].checksum = body.meta.checksum
16  })
17  // OR
18  return {
```

```

19     checksum: body.meta.checksum
20   }
21 }

```

## 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

```

1  // Allowed
2  [
3    {
4      id: '1234',
5      title: 'atitle',
6      description: 'adescription'
7    }
8  ]
9  // JavaScript can parse JSON without any special/unstandard logic, so this is still valid
   separation.
10 '[{"id":"1234","title":"atitle","description":"adescription"}]'
11 [
12   id: '1234',
13   titleAndDescription: 'atitle-adescription'
14 ]
15
16
17
18 // Disallowed
19 // Violation of guideline #1
20 ['1234,5678,9101112', 'onetitle,twotitle,threetitle', 'adescription,anotherdescription']

```

```

21 [
22   ['1234', 'atitle', 'description'] // Violation of guideline #2
23 ]
24 [
25   {
26     // Violation of guideline #3
27     id: 'm', // ID is encoded differently from the parent store
28     title: 'atitle',
29     description: 'adescription'
30   }
31 ]

```

## 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

```

1 // Allowed
2 export const resources = create()
3 export const derivedResources = createDerived()
4
5 export const resources = create()
6 export const derivedResources = createDerived()
7 export default resources
8
9 // Allowed, but semantically flawed
10 export const resources = create()
11 export const derivedResources = createDerived()
12 export default derivedResources
13
14 // Disallowed
15 export const resources = create()
16 export const derivedResources = createDerived()
17 export default create()

```

## 6 Caching of Encrypted Resources

This section is to be written in the next draft of this document.

## 7 Application of this Document

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

### 7.1 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 L<sup>A</sup>T<sub>E</sub>X, anyone that would like to submit changes, issues, or suggestions is encouraged to do so at <https://github.com/very-amused/SMSX>.