State Management Standard X

Keith Scroggs

24 August 2020

Contents

Code Listings

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 encouraged practices will be efficient with the usage of available computer resources, often allowing for it to be run with more acceptable speed on lower performance computers.

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 Stores, in order to write more consistent code that follows the principles stated above.

1.1 The Power Saw Problem

The obtainment of these principles is not impossible or even difficult using unopinionated state management tools such as the ones listed in the previous section, but because these tools are so general, vast inconsistencies exist between state management techniques. Different programs may have radically different approaches to the handling of state, which creates an unnecessary challenge for programmers joining the development of such a system. Another resulting problem of this unopinionated nature is that unpredictable, inefficient code and logic is allowed and not warned against by any means. As an analogy, it is understandable that with a tool as simple and powerful as a power saw, it is very easy for a woodworker to make not cuts that are not straight, may contain jagged edges, and will ultimately hinder the final product. The saw itself will not warn against making these cuts, which is why the woodworker may desire a guide rail, to establish consistently straight and clean cuts. This standard aims to act as such a guide rail for client-side state management logic.

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.

This document will include code listings in JavaScript pseudocode (based on Svelte Stores) to provide example implementations following this standard. In each of these listings, any text following two forward slash characters (//) is to be interpreted as a comment. Any text meeting this definition DOES NOT alter the behavior of the code, it serves as a form of explaining one or more parts of the listing.

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 valid UTF-8 strings.

3. Can be exported as a JSON encoded string with the use of no more than two non-custom transformations (must be included in the language).

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.

Listing 1: Maps

```
1   // Allowed
2   {
3       id: '123',
4       anotherKey: 'anotherValue'
5   }
6
7   // Disallowed (all keys must be UTF-8 strings)
8   {
9       1: 'aValue',
10      2: 'anotherValue'
11  }
12  // Disallowed (cannot be exported as JSON with the use of no more than two non-custom
        transformations)
13  '-id: "123"; -anotherKey: "anotherValue"'
```

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

Listing 2: ID

```
1   // Allowed
2   '1234'
3   '1234abcd'
4
5   // Disallowed (all IDs must be strings)
6   1234
7   true
8   false
9   // Disallowed (all IDs must be unique)
10  '12345'
11  '12345'
```

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 store used to manage a singular resource or piece of state (i.e user authentication/identity) is only managed by the section on single resource stores. Any other guidelines in this document are NOT REQUIRED for single resource stores. For more detail and examples of resource structure, see subsections 3.1 and 3.2.

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 one or more sources 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. A parent data source MAY be either client-side or server-side. A store MAY have more than one parent data source. If a server-side 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 parent data source SHOULD be nonvolatile in itself, meaning that it is not entirely stored in memory. Both of these terms refer exclusively to the stores themselves, not any parent data source(s).

Passing refers to when a method returns immediately when called, without performing any interactions or operations on any data or throwing/returning any form of error. In simpler terms, this is when a method does nothing, returns nothing, and throws nothing.

Listing 3: Passing

```
1  // Example of an init method passing
2  async init() {
3      return
4  }
5
6  // Certain languages may allow an empty block of code, may require a return statement, or
       may require a key word to indicate an empty block of code (such as Python's pass key
       word, where this term originates from)
```

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 would be exported as `[object Object]`. This standard allows for such a loss of information to occur during an export, at the benefit of allowing complex class instances to exist in stores without the usage of methods of encoding that may result in performance loss (such as the previously stated CryptoKey example).

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 MAY be duplicated between multiple stores, as long as the resource's ID is the same in all instances, and is still guaranteed to be unique in each store.

If a store is nonvolatile, its resources MUST be identical in structure to their parent data source. This ensures that when committing changes in the resource to the parent data source, no custom logic (a point where unpredictable behavior may be introduced) is required.

Listing 4: Valid Resource Structure

```
1  // Where the parent data source returns this resource
2  '123': {
3      title: String,
4      description: String,
5      meta: {
6          cryptoKey: CryptoKey,
7          index: Number
8      }
9  }
10
11 // Allowed
12 '123': { // Where '123' is a unique ID
13     title: String,
14     description: String,
15     meta: {
16         index: Number,
```

```
17              key: CryptoKey
18          }
19  }
20  '123': {
21      id: '123'
22      title: String,
23      description: String,
24      meta: {
25          index: Number,
26          key: CryptoKey
27      }
28  }
29
30  // Disallowed (resource structure must be identical to any parent data sources)
31  '123': {
32      id: '123'
33      title: String,
34      description: String,
35      index: Number,
36      key: CryptoKey
37  }
38  // Disallowed (the ID key used to access a resource must be a string)
39  true: {
40  }
41  // Disallowed (if the ID is included as a property, that property must be named id)
42  '123': {
43      wrongMember: '123'
44  }
45  // Disallowed (each resource must be a map)
46  '123': []
```

3.2 Single Resource Stores

Not all stores are used to manage multiple resources by ID. Sometimes, a store is necessary simply to manage a single resource or piece of state, this is referred to as a **single resource store**. A common example of this is that a store may be used to tell components of a program if a user is authenticated. ***The only requirement of a store of this nature is that the data component is a map.*** No other guidelines or requirements outlined in this standard need to be followed for a single resource store. Greater freedom is granted to the programmer for single resource stores, as their functionality and design can vary greatly by their function in a program.

3.3 Store Structure

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 5: Store Structure: 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 a good practice that results
         in more specific imports
15   export const users = create()
16
17   export default users
```

3.4   Store Data

A store's data MUST be a map. 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.

3.5   Store Methods

A store MUST contain the methods defined in subsection 4.1. These methods MUST behave as defined in subsection 4.2.

3.6   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 any data subscribed to 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, and any async functionality MUST be awaited. This is to ensure that any rejected promises are properly passed upstream so that they may be handled. Any event driven API accessed within a method MUST be either wrapped as a promise and awaited, or have an event listener placed on the error event to throw an error, therefore rejecting the method itself. If the event provides an error, the method MUST NOT intercept or alter this error in any way. If it is possible to wrap an event driven API in promises, this SHOULD be done to encourage simpler and more easily understandable code.

```
1    // Correct
2    async init() {
3        await addToStore('myStoreName', {
4            id: '123',
5            someProperty: 'someValue'
6        })
7    }
8    // Correct (although promises should be preferred when available)
9    init() {
10       const req = someIDBstore.add({
11           id: '123',
12           someProperty: 'someValue'
13       })
14
15       req.onerror = () => {
16           throw new Error(req.error)
17       }
18   }
19
20   // Incorrect (promise is not awaited)
21   async init() {
22       addToStore('myStoreName', {
23           id: '123',
24           someProperty: 'someValue'
25       })
26   }
27   // Incorrect (no handling for the error event)
28   init() {
29       const req = someIDBstore.add({
30           id: '123',
31           someProperty: 'someValue'
32       })
33   }
```

Listing 1: Test Listing

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 of 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 map 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 MUST 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 map containing 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 6: 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 map 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 7: Update Method

```
1  update(id, data) {
2      // Looks weird, but this is valid thanks to namespacing
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 8: Delete Method

```
async delete(id) {
    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')
        // This method can choose to fail silently
        return
    }

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

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 9: 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
        })
    }
}
```

4.7 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 10: 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 map. In fact, a primary use case of derived stores is to transform a store from a map to another format (such as an array). If a derived store is not in the format of a map, 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 map, 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 property 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 11: 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 (custom separators are not allowed)
19  ['1234,5678,9101112', 'onetitle,twotitle,threetitle', 'adescription,anotherdescription']
20  // Disallowed (IDs are not preserved as child members named ID)
21  [
22      ['1234', 'atitle', 'description'] // Violation of guideline #2
23  ]
24  // Disallowed (the different encoding of the ID from the parent store makes it impossible to
         retrieve the resource from the parent store using the information given in this derived
         store)
25  [
26      {
27          id: 'm',
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 12: 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 contains guidelines for the handling of zero-access encryption in compliance with this standard.

6.1   Overview

Decrypting information on the client-side requires a nontrivial amount of resources and time, and SHOULD be minimized as much as possible. In order to accomplish this, checksums SHOULD be used in a specific way to ensure that store methods avoid decrypting information that it already has a copy of.

6.2   Calculating Checksums

Checksums MUST be taken from resources on the server-side, before being sent to the client. This is both an additional effort towards minimizing client resource usage, and a preventative measure against errors/inconsistencies in client code for taking checksums. Checksums MUST be taken from the encrypted form of a resource, NOT the decrypted form, as this would undermine the entire point of avoiding unnecessary decryption operations. The checksum MUST be taken from a BINARY, CONCATENATED form of the resource being retrieved. If the resource is stored in binary and then converted to another format before being sent to the user (such as hex or base64), all properties MUST be concatenated while still in binary form. The concatenation order of these properties does not matter, as long as it is consistent between each time a checksum for the resource is calculated. Each time the checksum is calculated for a resource, it MUST be identical, unless the resource itself has changed.

6.3   Checksum Algorithms

The hash used to calculate checksums DOES NOT need to be cryptographically secure, as it is only ever taken of data that is already encrypted, and only used to validate whether a change has happened to that form of the data. While security is not an important function of this hash, speed is. Older, faster hashes such as MD5 and SHA-1 are preferred to newer hashes that provide greater security at the cost of speed.

6.4 Cached Storage

When storing a nontrivial amount of data, it is important that clients have a fast, indexed method of storage and retrieval. On the web, IndexedDB SHOULD be preferred to localstorage for storing cached resources and checksums.

6.5 Applying Checksums during Initialization

A client SHOULD perform logic to avoid unnecessary decryption during a store's `init` method as follows:

1. The resource is retrieved from a parent data source, the retrieved data includes a checksum for the resource's encrypted form.

2. The client tries to retrieve a cached copy of the resource from its cache.

3. If retrieving a cached copy of the resource fails, the client decrypts the resource, and stores it along with its checksum in the cache.

4. If a cached copy is retrieved, the client compares the checksum of the cached copy with the checksum retrieved from the parent data source.

5. If the two checksums match, the client knows that the resource has not changed, and adds the cached copy of the resource to the store.

6. If the two checksums do not match, the client knows that the resource has updated, and decrypts the resource. The client then adds this resource to the store, and updates its cache to contain the updated data and checksum.

This logic DOES NOT alter or otherwise take precedent over the required methods for updating application state specified in section 4, it simply provides a design pattern to greatly speed up the `init` method when dealing with encrypted data.

Listing 13: Retrieving Encrypted Resources

```typescript
type KeyedObject = {
    id: string,
    [index: string]: unknown
}
// Retrieve an IDB record by id
async function getByKey(storeName: string, key: string): Promise<KeyedObject>

// Update or create an IDB record, must include id
async function updateWithKey(storeName: string, data: KeyedObject): Promise<void>

// Decrypt the only encrypted parts of a resource (real code would require actual resource-
    specified logic to accomplish this)
async function decrypt(data: KeyedObject): Promise<KeyedObject>

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

    for (const resource of body) {
        // If a cached version with matching checksum exists, add that version to the store
            and continue
        const cached = getByKey('resources', resource.id)
        if (cached && cached.checksum === resource.meta.checksum) {
            update((store) => {
                store[resource.id] = cached
                return store
            })
            continue
        }

        // At this point, we can decrypt (and flatten) the resource, knowing that it isn't a
            wasteful operation
        const decrypted = {
            ...await decrypt(resource),
            id: resource.id,
            checksum: resource.meta.checksum
        }

        update((store) => {
            store[decrypted.id] = decrypted
            return store
        })

        // Update the cache to contain the newly-decrypted version of the resource
        await updateWithKey('resources', decrypted)
    }
}
```

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 LaTeX, anyone that would like to submit changes, issues, or suggestions is encouraged to do so at https://github.com/very-amused/SMSX.