Analyzing and implementing a third-party state machine library for FriendlyReader and TeCST

David Holmstedt

Supervisor: Arne Jönsson
Examiner: Robert Eklund
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstakta kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

For ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© David Holmstedt
Abstract

FriendlyReader and TeCST are text simplification tools developed at Linköping University, using an API service at the university called SAPIS. Both tools are web services that run in the browser for users. To improve the services an implementation of state-transition tracking was chosen as a way to both improve the website by enabling the user to undo actions, but also enable the client to cache information which lowers the amount of requests required to SAPIS. Two libraries, called MobX and Redux, where compared to find the one which worked best for FriendlyReader and TeCST. The main difference between MobX and Redux is the programming paradigms, MobX is object-oriented while Redux is functional. In the end MobX was chosen due to the object-oriented nature of the library which is more familiar for beginners while also requiring less code to achieve similar results. MobX lacks native support to keep track of previous state transitions which is required to be able to go back to a previous state. Using MobX a new library called GlobalStore was created to solve this problem. An implementation example for synonyms was produced as a proof of concept for FriendlyReader.

Keywords: MobX, Redux, FriendlyReader, TeCST, SAPIS, state-machine, state-transition
Acknowledgments

I want to thank my supervisor Arne Jönsson for continued support and the chance to create something with practical uses from scratch. Thanks to my fiancé Rebecka Lundin for being supportive and helping me find the strength to finish. I also want to thank all members of my thesis group for the continued feedback. Lastly thank you to my cat for keeping me sane during the late nights.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vii
List of Tables viii

1 Introduction 1
1.1 Purpose ................................................. 1
1.2 Developers / End users ................................. 1
1.3 Thesis aims ............................................. 1
1.4 Delimitations ............................................ 2

2 Background 3
2.1 Current implementation of FriendlyReader and TeCST services ......................... 3

3 Comparing Redux and MobX 8
3.1 States and Transitions ................................. 8
3.2 Native JavaScript 9
3.2.1 jQuery ............................................... 9
3.3 Redux .................................................. 9
3.3.1 Actions .............................................. 9
3.3.2 Reducers ............................................ 9
3.3.3 Store ............................................... 10
3.3.4 Using Redux ......................................... 10
3.3.5 Ability to debug ..................................... 11
3.3.6 Developer tools ..................................... 11
3.3.7 Summary ........................................... 12
3.4 MobX ................................................ 12
3.4.1 Observable .......................................... 13
3.4.2 Reactions (Autorun / (Observe), When, Reaction) ............................. 14
3.4.3 Computed values ..................................... 14
3.4.4 Action ............................................... 15
3.4.5 Ability to debug ..................................... 15
3.4.6 Summary ............................................ 16
3.5 NodeJS and webpack .................................... 16
3.5.1 NodeJS .............................................. 16
3.5.2 webpack ........................................... 16
3.6 Breaking down TeCST and FriendlyReader ................................................. 16
3.6.1 SAPIS .............................................. 16
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Front page of FriendlyReader</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Analyzed text and Synonym suggestion for the word “ett”</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>TeCST frontpage and the dialog for the different options in the feedback option</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Suggestions for simplification of sentences can be seen to the right</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Example code from reduxcoreconcepts</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Action Reducer example code from reduxcoreconcepts</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>Main Reducer example code from reduxcoreconcepts</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>Usage example example code from reduxstore</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>The user interface for the Redux Dev-tools for Google Chrome and Mozilla Firefox</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>A todo list class structure example mobxmain</td>
<td>13</td>
</tr>
<tr>
<td>3.7</td>
<td>Flowchart showing the way MobX reacts to actions mobxmain</td>
<td>13</td>
</tr>
<tr>
<td>3.8</td>
<td>an example how a generic autorun reaction is defined from mobxautorun</td>
<td>14</td>
</tr>
<tr>
<td>3.9</td>
<td>an example of a when reaction from mobxwhen</td>
<td>14</td>
</tr>
<tr>
<td>3.10</td>
<td>an example of a reaction reaction</td>
<td>14</td>
</tr>
<tr>
<td>3.11</td>
<td>A computed value has a set reaction to execute when changed mobxcomputed</td>
<td>15</td>
</tr>
<tr>
<td>3.12</td>
<td>An example class where an action increases the tick increment every second mobxaction</td>
<td>15</td>
</tr>
<tr>
<td>3.13</td>
<td>The request message to SAPIS</td>
<td>17</td>
</tr>
<tr>
<td>3.14</td>
<td>The request options to SAPIS</td>
<td>17</td>
</tr>
<tr>
<td>3.15</td>
<td>A response received by SAPIS</td>
<td>17</td>
</tr>
<tr>
<td>3.16</td>
<td>A response received by SAPIS</td>
<td>18</td>
</tr>
<tr>
<td>3.17</td>
<td>Impure and pure function</td>
<td>20</td>
</tr>
<tr>
<td>3.18</td>
<td>A pure function example for Redux robinwieruchconfusion</td>
<td>20</td>
</tr>
<tr>
<td>3.19</td>
<td>A comparative example of MobX and Redux robinwieruchconfusion</td>
<td>22</td>
</tr>
<tr>
<td>3.20</td>
<td>An illustration of the learning curves of MobX and Redux youtubemobxvsredux</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>UML class diagram of the GlobalStore library</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>UML-diagram of Synonym implementation</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Selecting a synonym and a dropdown with the top three choices for the Swedish word “vid”</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>The synonym changed by the user to “samman”</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Code stripped of all user interface code to illustrate the use of GlobalStore</td>
<td>30</td>
</tr>
</tbody>
</table>
List of Tables

3.1 A state is changed into a new state through a transition. ................. 8
1 Introduction

The Bachelor thesis will explore different libraries used to implement state machine patterns to optimize server traffic and expand two web services that Linköping University (LiU) has created called TeCST and FriendlyReader. FriendlyReader is developed to help people with difficulty reading and understanding text while TeCST is made for writing articles. They achieve this functionality by connecting to another service developed by LiU called SAPIS which provide advanced language technology modules to the user. Both of these allow the user to input a text and then provides different tools to help make the text easier to comprehend such as summarizing, replacing sentences and suggesting synonyms for individual words in the text. State management will allow the users to more freely explore the different options without having to worry about non revertible changes and give the services more potential to help people with language difficulties.

1.1 Purpose

Implementing state machines in of itself can be a difficult task but there exists several libraries on the market to alleviate, this thesis will explore the most popular state management libraries for JavaScript web development and identify the library that fits the current users at LiU, while also being relatively easy to use and for new users to grasp. The library will be generic enough to allow it to be applied both TeCST and FriendlyReader without major modification. Having a history for the API calls is important for two reasons, it will make the experience on the site more intuitive by being able to undo changes which also provides smoother changes for already cached results. The second reason is on the server side, a local client history will reduce the API load by eliminating unnecessary API calls which will provide a more stable service.

1.2 Developers / End users

The library will be developed as a generic library to be used in both FriendlyReader, TeCST and other potential rewrites or new services developed in the future in relation to these tools. This creates a few requirements on the library for it to be both easy to understand and easy to use while still being flexible enough to be useful in a multitude of usecases. The requirements has the potential to have a large impact on how the library has to be made.

1.3 Thesis aims

Below are the identified aims for the thesis:

- Explore and compare different libraries that implement state management.
- Implement a state management using an external library that is easy to use even for less experienced developers which can be integrated into FriendlyReader and TeCST.
1.4 Delimitations

This thesis will only focus on the underlying data under the hood of the program and not handle a fully implemented UI into FriendlyReader and TeCST, however an example will be made to show how the underlying libraries work as developer documentation.
2 Background

TeCST and FriendlyReader are two services that was developed by LiU and are in use today and can be accessed in a web browser.

2.1 Current implementation of FriendlyReader and TeCST services

FriendlyReader is a service that aim to alleviate issues with reading impairment such as dyslexia (Sandström, 2012). TeCST is a tool for editors that implement the functions available in FriendlyReader, instead of automatically simplifying the text like FriendlyReader it provides suggestions to the editor on how to change the text. This allows editors to be more inclusive of people with reading impairment through the use of a simple tool.

Both pages are implemented in similar ways due to their overlapping toolkit and they both use use HTML and JavaScript to create a reactive experience where different things progress depending on user input. They also use an API service provided by LiU called SAPIS which contains all the text simplification tools that FriendlyReader and TeCST both use.

In Figure 2.1 we can see the current interface for FriendlyReader, it allows the user to input text and then a run button which analyses the text and presents it in a similar way as Figure 2.2. Here the user can change synonyms and receive tools such as read out loud and reading rulers.

The current versions of FriendlyReader and TeCST have a similar structure on how they fetch information from the backend by using an ajax command through the jQuery library. Because they both use the same backend their requests to the backend are very similar which makes it easier to create a single solution for the two services. Nevertheless they are also different in a few ways. FriendlyReader is more static and does not allow the user to change the settings sent to SAPIS.

Figure 2.3 shows the main page of TeCST, the most prominent difference from FriendlyReader is that TeCST allows the user to edit the text after analysis and provides suggestions about the text in information tabs to the right. In Figure 2.4 the text simplification tab is opened to the right and it shows suggestions on how to change sentences and also the reason for the change.

TeCST use more libraries for different things such as D3 for diagrams and ckeditor for the webpage editor. These libraries have no real impact on the work in this report and is outside of the libraries scope because they handle UI elements and will not be a focus for the development of the library.

Most of the functionality in TeCST and FriendlyReader comes from the SAPIS interface and UI rather than logic contained in the webpage.
2.1. Current implementation of FriendlyReader and TeCST services

Figure 2.1: Front page of FriendlyReader.
2.1. Current implementation of FriendlyReader and TeCST services

Figure 2.2: Analyzed text and Synonym suggestion for the word “ett”.
2.1. Current implementation of FriendlyReader and TeCST services

Figure 2.3: TeCST frontpage and the dialog for the different options in the feedback option.
2.1. Current implementation of FriendlyReader and TeCST services

Figure 2.4: Suggestions for simplification of sentences can be seen to the right.
3 Comparing Redux and MobX

There are a lot of libraries to choose from for state management in JavaScript, there are however two libraries that are used extensively by the web development market which are Redux and MobX (State Management Tools Results, 2017).

3.1 States and Transitions

A variable in a program contains some information that one wants to keep track of, which in turn can be read and/or changed by the program/user. The variable is always in its “current state” and when it is changed a transition into a new state is performed. This means that potentially every action in a program can be logged as a discrete state. A state could then be seen as an interval in time through the change of data. In Table 3.1 a number is used as an example to illustrate a state.

<table>
<thead>
<tr>
<th>State 1</th>
<th>Transition</th>
<th>State 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer: 1</td>
<td>Add 4</td>
<td>Integer: 5</td>
</tr>
</tbody>
</table>

In the example in Table 3.1 we have two discrete states of 1 and 2 where the transition changes the value. The states are still the same variable but we lose the information about the previous state by changing the underlying variable. By keeping track of these we can use the chain of transitions in practice turn back time of the state. This does come with a cost to the system where more memory is needed to keep track of all the values, but it is not required to save the entire variable in every state to achieve this result. Instead we optimize the memory usage by only keeping track of the transitions in the data rather than keeping track of the entire value of the variable. Keeping track of state transitions is a normal problem in software development and this means that there are libraries that have implemented functions to make it easier.

3.2 Native JavaScript

JavaScript is an asynchronous language where you do not specify data type but instead all variables are objects that can take on any value like list, array, integer and string. This makes the language very mutable which makes keeping track of states a potential issue because the data type is potentially unknown, there are a few different libraries that try and tackle this problem by allowing variables to be defined and monitored for potential change to make the state mutations predictable (Motivation, 2018). Another way to solve the problem with keeping track of state transitions is to use external libraries for JavaScript where others already solved the state mutability problem. Two notable examples are Redux and MobX.
3.2.1 jQuery

There are ways to extend the functionality of Native JavaScript. One way is through jQuery which is a JavaScript library with a set of helper functions that makes JavaScript easier to use by abstract away some of the more advanced code required for functions (About jQuery, 2015). An example is $\$(\#id\$) to get a DOM element on a webpage and more advanced functions such as removing the requirement to use XMLHttpRequests for cross-site communication (jQuery Introduction, n.d.). FriendlyReader and TeCST use jQuery in the current implementation to be able to use an ajax request to request the information from SAPIS which is a service run on LiU servers.

3.3 Redux

One of these libraries that try and handle state mutations is Redux which is a library for NodeJS and JavaScript. Redux defines the states of a transition through explicit definitions as a plain object in JavaScript. These objects are then changed by actions that use the payloads of information and it is the only way to move information in and out of the Redux store (Core Concepts, 2019). Reducers are then used to actually change the applications state in response to the actions. It achieves this by using a functional programming design paradigm which is different from the more common object-oriented design where it achieves new states pure functions and immutable objects (Wieruch, 2017a).

3.3.1 Actions

Actions are the actual payload that the site sends to Redux, here the type and data is defined. An action is defined as a JavaScript object such as this:

```
1 { type: 'ADD_TODO', text: 'Go to swimming pool' }
```

Figure 3.1: Example code from Core Concepts (2019).

Using Figure 3.1 as example the type “ADD_TODO” is the type of action and “Go to swimming pool” is the payload/data that the action contains. The action by itself does not actually do anything at this point but is instead a container of information that is then used by the reducer (Core Concepts, 2019).

3.3.2 Reducers

To tie the actions and the state together Redux uses something they call a reducer. A reducer is where the actual information is used and inserted into the state and then it returns the next state in the application which now contain the data from an Action. A reducer can be seen in Figure 3.2 by using the action provided to the function the next state is generated by the reducer. The actual code is not important but more importantly the type is used to decide what will happen.

However this is still only one reducer for a number of different actions for the reducer. Usually there are multiple different actions that can fire at any given time and they are not necessarily linked or handle the same type of data. Because of this we need to actually manage the entire state of the app by using a main reducer that calls the sub reducers such as the example above.
3.3. Redux

```javascript
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map((todo, index) =>
        action.index === index
        ? { text: todo.text, completed: !todo.completed }
        : todo
      )
    default:
      return state
  }

  function todoApp(state = {}, action) {
    return {
      todos: todos(state.todos, action),
      visibilityFilter: visibilityFilter(state.visibilityFilter, action)
    }

  function visibilityFilter(state, action) {
    // Filter based on action type
  }

  return todoApp()
}

Figure 3.2: Action Reducer example code from Core Concepts [2019].

Figure 3.3 is a function that combines two functions that handle different actions. In this example `todos` handles insert and removal while `visibilityFilter` handles showing and hiding of todo parts.

3.3.3 Store

The store is where the state is actually stored. The only way to change the current state in Redux is through an action and the state itself is read only which means that you cannot assign values directly to the state. This is to ensure that views or network callbacks will never directly write to the state but instead try and transform it through an action.

3.3.4 Using Redux

Up until this point all definitions are just normal JavaScript objects that can be made without Redux. Figure 3.4 shows how everything is fitted together by using Redux to store the states and actions through reducers to actually change it.

We first import the defined functions and Redux store. We then define the actual store by using `const store = createStore(todoApp),` this creates the data structures by using the definitions to allow different actions to change the underlying state. Then by using `store.dispatch` we can change the data inside the store object by using the Redux store which then can be viewed through the `store.getState()` function.
3.3. Redux

Usage example (https://redux.js.org/basics/store)

1 // Import Redux Store functionality
2 import { createStore } from 'redux'
3 // Import the previously defined reducers and actions
4 import todoApp from './reducers'
5 import {
6   addTodo,
7   toggleTodo,
8   setVisibilityFilter,
9   VisibilityFilters
10 } from './actions'
11
12 // Create the actual store
13 const store = createStore(todoApp)
14
15 // Log the initial state
16 console.log(store.getState())
17
18 const unsubscribe = store.subscribe(() => console.log(store.getState()))
19
20 // Dispatch some actions
21 store.dispatch(addTodo('Learn about actions'))
22 store.dispatch(addTodo('Learn about reducers'))
23 store.dispatch(addTodo('Learn about store'))
24 store.dispatch(toggleTodo(0))
25 store.dispatch(toggleTodo(1))
26 store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED))
27
28 // Stop listening to state updates
29 unsubscribe()

Figure 3.4: Usage example example code from Store (2018).

3.3.5 Ability to debug

The ability to easily debug Redux is one of its main features. The explicit way of defining actions, reducers and separated stores gives it a logical chain of events in the state transitions which is easy to follow. This gives the state transitions a larger predictability which enables Redux to provide a strong developer tool. However the code itself can be hard to read and understand due to the way Redux combines actions and reducers into a single JavaScript object which are then ran inside a specific storage variable where actions are dispatched. While this structure can be improved by good coding standards it is not required in Redux which can potentially make a larger project difficult to understand (Kasireddy, 2017).

3.3.6 Developer tools

While not entirely necessary for the development in Redux it comes with a fully-fledged developer tool for the web-browsers Mozilla Firefox, Google Chrome, Electron, etc. but also a set of tools for non-browser environments Johnston (2019).

The Redux tools allows the user to see the states changes. In Figure 3.5 the initial state for the application is shown as @@INIT to the left and each state transition is listed after. On the right hand side it shows the data pertaining to that state change.
Figure 3.5: The user interface for the Redux Dev-tools for Google Chrome and Mozilla Firefox.

This tool enables the developer to go back and forwards in time through the states via a visual interface which provides an intuitive way to understand how internal state changes.

3.3.7 Summary
Redux is a very explicit way of handling states and their transitions. Every action is explicitly defined through JavaScript objects which are then fed into reducers which can run any logic relating to the action payload. This creates a way to represent state transitions through data structures but it also requires all the wanted actions to be explicitly defined which creates large definition databases through logical data structures and very little behind the scenes processing is done (Kasireddy, 2017).

3.4 MobX
Another library that handles transitions is called MobX. It tries to make state management simple and scalable by using functional reactive programming (FTRP). MobX themselves describes their philosophy as follows “Anything that can be derived from the application state, should be derived. Automatically. Which includes the UI, data serialization, server communication, etc.” (MobX, 2017). MobX handles data in an implicit way, where compared to defining specific structures about objects explicitly MobX instead specifies a specific object to track for state change regardless of the actual structure of the object. It does this by making objects observable to MobX through tagging of the JavaScript variable. Each time the variable is changed it goes through an action which is tagged with MobX in the same way as an observable. These tags are called decorators by MobX and include observable, action and computed(value).

Figure 3.6 is a small example of a split problem where the todo is a single object containing information and the TodoList contains actions to modify the todo objects in a list. By breaking up the problem into modules the program is more easily debugged. Modules are not a MobX solution and therefore relies entirely on the programmer doing a correct structure. This clearly shows that MobX follows an object-oriented programming paradigm (Wieruch, 2017a).
3.4. MobX

```javascript
class Todo {
    constructor() {
        id = Math.random();
        title = "";
        finished = false;
    }
}

decorate(Todo, {
    title: observable,
    finished: observable
});

class TodoList {
    constructor() {
        todos = [];
    }
    addTodo(text) {
        todos.push(new Todo(text));
        ... etc ...
    }
}

decorate(Todo, {
    todos: observable,
    addTodo: action
});
```

Figure 3.6: A todo list class structure example MobX (2017).

### 3.4.1 Observable

The main functionality in MobX comes through their observables. An observable is a state that MobX will track and allows the developer to attach functions that get executed when the value of an observable is changed which MobX calls Reactions (not to be confused with React the UI library). An observable is just a simple JavaScript variable that has been decorated as an observable through the MobX library. To understand the impact these observables have on state transitions we need to look at the different ways these state transitions can be reacted to through MobX. A flowchart can be seen in Figure 3.7 of how an action is propagated through the MobX library.

![Flowchart showing the way MobX reacts to actions](image)

Figure 3.7: Flowchart showing the way MobX reacts to actions MobX (2017)
3.4.2 Reactions (Autorun / (Observe), When, Reaction)

Autorun, When and Observe is used to create a reactive function when the underlying value of an observed variable is changed or read (in the case of observe). There are some differences between these. Figure 3.8 illustrates the Autorun syntax which is executed when a value changes in an observed variable but will always execute on any change to any observed variable. Observe is a variation to Autorun used with React (The UI library) but is mentioned for due diligence (Autorun, 2017).

```javascript
@autobind
var title = "MobX is hard"

//Specify the reaction
reaction => console.log("Title changed to ", title)

//Change the variable
//This would execute the reaction and will print "Title changed to MobX is easy"
title = "MobX is easy"
```

Figure 3.8: an example how a generic autorun reaction is defined from Autorun (2017).

When, just like Autorun, will execute on a change but it instead observes and runs a predicate which executes the effect when the predicate returns true. Figure 3.9 will execute if !this.isVisible resolves to true, which will subsequently run this.dispose.

```javascript
// If this equals true execute next row
() => !this.isVisible, // If this is true, dispose this.dispose()
// true happened, dispose object
)`

Figure 3.9: an example of a when reaction from when (2017).

Last of the reactions in MobX is the reaction which can be seen in Figure 3.10. Reaction also executes on value change but differs in that you can specify which specific observable variable it should react to.

```javascript
//@observable
var title = "MobX is hard"

//Specify the reaction
reaction => console.log("Title changed to ", title)

//Change the variable
//This would execute the reaction and will print "Title changed to MobX is easy"
title = "MobX is easy"
```

Figure 3.10: an example of a reaction reaction.

3.4.3 Computed values

Computed values which are annotated with computed is similar to a formula in a spreadsheet program like excel. When the value is read it computes the value and returns the computed result. A computed value can be used as a variable which allows a function to be reacted to. What this allows for is more advanced values being calculated through more simple ones as seen in Figure 3.11 where a computed function has a reaction set to the value change of squared instead of creating a state variable that contains the squared value of length as well.
class Foo {
    constructor() {
        reaction (() => this.squared,
            () => { 
                Console.log("Value of computed value squared changed"); 
            }) ;
    }
    @observable length = 2;
    @computed get squared () {
        return this.length * this.length;
    }
    set squared (value) {  
        this.length = Math.sqrt(value);
    }
}

Figure 3.11: A computed value has a set reaction to execute when changed (@)computed (2017).

3.4.4 Action

The observations are the state of data and actions are anything that modifies the state, nothing stops the developer from directly modifying an observed variable (this can be configured in MobX but isn’t default) reactions will still function. However MobX allows functions to be decorated as an action, actions should only and always be used on functions that modify a state. Functions that only read the date should not be marked as action. Figure 3.12 shows how increment is set to an action through @action.

class Ticker {
    @observable tick = 0
    @action
    increment () {
        this.tick++ // 'this' will always be correct
    }
}

const ticker = new Ticker()
setInterval(ticker.increment, 1000)

Figure 3.12: An example class where an action increases the tick increment every second action (2017).

3.4.5 Ability to debug

MobX uses tagging to define non explicit variables which means that the data type and structure can change at any time either through direct assignment or actions. This creates a problem where the implicit nature can make it problematic to debug where it is up to the developer to do it correctly and mistakes can be missed because implicitly there is nothing wrong (Kasireddy, 2017). There are ways to make this more explicit by configuring MobX to only allow state change through actions, but it is not default behavior of MobX (action, 2017). However the way MobX is developed gives it a modularity where it forces you to adapt.
an object oriented approach through a class structure alleviates this problem by forcing the developer to break up larger problems into smaller solutions.

### 3.4.6 Summary

MobX is a library where states are implicitly stated and distributed over different classes and each observed variable tracks state changes individually. The implicit nature of MobX makes it very flexible where data structures are not explicitly stated and does not require the developer to create specific transitions. This requires the developer to be more aware how their states will change and consider the possibility of them being changed in an unusual manner both through actions and through direct assignment and it does not restrict the change because no data structure is explicitly stated. As said earlier (reference to actual chapter) this has an effect on how easily it can be debugged, but requires less code for a similar state transition that is explicitly stated. MobX can however be configured to allow only state changes to happen through actions to make it require more explicit definitions of state transition functions but as previously mentioned this is not default.

### 3.5 NodeJS and webpack

Both MobX and Redux are written for NodeJS development which is extensively used in web development.

#### 3.5.1 NodeJS

NodeJS is a JavaScript runtime that is used to build scalable network applications that allows JavaScript to be run on the server which is classically a client side language (Nodejs Tutorial, n.d.). NodeJS is also used to build libraries to be used by webpages because of the larger support and availability of external libraries (About Nodejs, n.d.).

#### 3.5.2 webpack

NodeJS is designed to be run on the server but with a library called webpack (lowercase) which is a static module bundler. We can build these server-side files to static modules which are then loaded when the client starts. The reason for this is that we want to always run the state transition logging on the client because keeping all the transitions on the server would require the server to be able to handle potentially an unlimited amount of users. By putting it on the client we can distribute the memory load while still being able to use NodeJS libraries (Concepts, n.d.).

### 3.6 Breaking down TeCST and FriendlyReader

One of the main goals of this thesis is to try and find a way to reduce unnecessary requests to the server by keeping previous states and requests, but also in the process try and make the program more concise and easier to work with. In this chapter the libraries Redux and MobX will be fleshed out and compared to find the one that meets the aims in the most complete way.

To be able to figure out what the final library needs to be able to handle we need to first break down TeCST and FriendlyReader; what are actually functions in the program themselves and what is based in data.

#### 3.6.1 SAPIS

Because both TeCST and FriendlyReader use the same underlying server calls to SAPIS they require the same calls even if they use them in somewhat different ways. TeCST is the more
3.6. Breaking down TeCST and FriendlyReader

fully featured of the two with settings for the server call being different depending on user-input. The server request itself is pretty simple in structure in the form of a JSON input object of `document` and `options` as seen in Figure 3.13.

```json
1 {
2   "options":"OPTIONS",
3   "document":"EXAMPLE TEXT"
4 }
```

Figure 3.13: The request message to SAPIS.

The `options` contain the different settings that SAPIS can handle and each option is separated by a backslash ‘\’. Figure 3.14 illustrates some of the potential options:

```plaintext
1 "Feedback(-svo_-_pass2act_-_prox_-_quoteInv_-_split)"
2 "LexicalMetrics()"
3 "SurfaceMetrics()"
4 "StructuralMetrics()"
5 "Synonyms"
```

Figure 3.14: The request options to SAPIS.

The options themselves are just a simple string of text which means that to add a setting one just concatenates it to the `options` string. The Feedback setting has sub-settings but are still merged to the string in the same way except that they are separated by space, however this changes nothing to the underlying request.

Next object is `document` which just contains the entire text that SAPIS will analyze as a string using the options defined.

After the request is sent the server will send back a response with a delay between the request and the response due to it being sent over network. The response will have different contents depending on the options in the initial request, what is returned is a JavaScript dictionary where the different data values are in named indexes. Each of the return index values do have a different structure because SAPIS uses several different programs to produce the different results depending on the options.

In Figure 3.15 the response when the option “Synonyms” is used can be seen.

```json
1 response = {
2   ["synonym_suggestions"] = {
3       ["hej"] = {
4         ["level":4.1, "synonym":"hejsan"],
5         ["level":4.0, "synonym":"tjena"]
6       }
7   }
8 }
```

Figure 3.15: A response received by SAPIS.

This is a dictionary list within the “synonym_suggestions” which contains a list of the initial words and synonyms for the word and how well SAPIS think the synonym would fit.
While other responses are not in the same nice tree structure but instead they all come in a flat structure such as Figure 3.16.

This means that the response can have a lot of different data structures depending on the options that are passed in the request. This means that the final library has to be able to handle the transitions with deep list structures as well as flat structures and handle the transitions between these different data types.

```
1   response = {
2       "1" = 0,
3       "2" = 1
4   }
```

Figure 3.16: A response received by SAPIS.

### 3.6.2 Text Modification

Both FriendlyReader and TeCST use text of an unknown length which is passed to the server when sending requests to SAPIS. TeCST also allows the user to change the text after initial input while FriendlyReader only allows the text to be changed through summarization or replacing words with synonyms. This means that FriendlyReader is more limited in how the text can be changed which is easier to track, TeCST on the other hand allows change in any location like you can in programs like Office Word. This creates a problem where a simple solution for FriendlyReader might not be enough for the more advanced user possibilities of TeCST, they do however both use a big string with all the text that is stored in a HTML textarea element. The difference between the two is in how the changes will be tracked in the code.

### 3.6.3 Common elements

Even though these two different parts of SAPIS are functionally different they still have the same underlying function in the code and can all be thought of as primary types such as Numbers, Strings, Lists and Dictionaries with functions using and changing this data to archive functions for the end user.

Currently the data is modified through functions and direct assignment, all these different types of changes need to be encapsulated with functions that capture these transitions in data in some way, there are several ways to accomplish this.

This would mean that to track the data changes the final library needs to be able to track these primitive types in JavaScript.

### 3.7 Standalone

For completion it is worth mentioning that all libraries in this report can be done in native JavaScript and there is no need to actually use libraries but doing it this way could potentially introduce a lot of problems into the codebase and would introduce unnecessary complexity as it would need to be maintained by LiU if future problems arise with the code. Choosing a library that fits the use case is better for this project as it reduces the code users would need to learn and the required developer skill, and it abstracts away code from LiU to a library (Matthew, 2018).
3.8 Redux and MobX

Both MobX and Redux are fully capable of completing the task at hand and are functionally similar but differ on the philosophy on what the library actually will be used for and how to achieve said result. Both libraries are run through NodeJS which makes them similar in that they both require similar development environment. However the end user for the library should not be required to setup the environment for NodeJS (even though it is relatively easy) to be able to use the library in their web development. To avoid requiring a NodeJS development environment and not require a NodeJS server to run the webpage we use something called webpack which, put simply, recompiles the NodeJS code to a standard importable JavaScript file. This does of course imply that if someone would want to change anything regarding the library in the future they would need to setup the development environment to be able to do this. This is not a normal use-case and would only happen when new logic is introduced into the GlobalStore state-machine (Mobx vs Redux, n.d.).

For completion it is worth mentioning that the reason NodeJS is used is due to the fact that the best libraries require NodeJS, but this is a non-issue because webpack converts NodeJS to regular JavaScript.

3.8.1 Real world users of MobX and Redux

While both libraries can handle similar things it is good to start looking at the different places where either MobX or Redux has been chosen. The real world examples of Redux are easy to find and have several big companies using their library for their state management. Some notable examples are Twitter’s mobile site, Wordpress new admin page and lastly Mozilla Firefox for their web development debugger which is used when inspecting webpage sources (Redux FAQ: Miscellaneous, n.d.).

MobX real world projects are also easy to find. Some of the bigger more notable are companies such as Coinbase the leading bitcoin brokerage platform, Bank of America which is a large bank in the United States of America and Amazon web services also known as Amazon AWS which is one of the largest on-demand computing platforms in the world (Hawkins, 2017).

This shows that MobX and Redux are both used in large scale projects which gives both MobX and Redux a great pedigree where large companies use the libraries in a production setting for state management. This shows that both MobX and Redux are powerful libraries and that they are applicable in large scale website production. This does not mean that it can’t be used in smaller projects as well, both provide impressive lists of smaller projects from the hobby project to medium scale companies (Eriksson, 2018; Hawkins, 2019).

The fact that they both are used in real world projects makes choosing which library that is best fitted for the task of introducing state-machines in FriendlyReader and TeCST harder. This is because they are both used extensively by large subsets of the web development world and a lot of the articles on the internet run the risk of being biased because personal choice and preferences can have a large impact on what library that is actually chosen. This does not mean that the libraries are identical, they both have niches they try and fill but also have a totally different design philosophy when developing for each library.

3.8.2 Coding style differences between of Redux and MobX

The main differences between Redux and MobX is actually the different programming principles they use. Redux uses functional programming principles (Wieruch, 2017b) which is unlike the normal Object-Oriented programming which is used by languages such as C# and Java. Functional programming introduces a few concepts such as data immutability and pure functions which will provide side-effect-free functions which makes it easier to maintain systems (TK, 2018).
To explain what a pure function is we need to know what an impure function is. In Figure 3.17 two functions are presented the one to the left is impure which means that it is not guaranteed that it will return the same output each time it is run because the variable pi can be changed from an outside source which has the effect that you cannot trust that the same input will result in the same output. The pure function however gets all the required data through its parameters which means that given the same input to the function we can expect the same output. This means that the functions are deterministic and won’t cause any observable side effects because it is all contained within the function (TK, 2018).

```javascript
1 //Impure function
2 var pi = 3.14
3 function calculateArea(radius){
4     return radius * radius * pi;
5 }
6 calculateArea(10) //returns 314
```

```javascript
1 //Pure function
2 var pi = 3.14
3 function calculateArea(radius, pi){
4     return radius * radius * pi;
5 }
6 calculateArea(10, pi) //returns 314
```

Figure 3.17: Impure and pure function.

The next part of functional programming is the immutability which means that you cannot change a value that has already been set, but instead you create a new object with the new value (TK, 2018).

With this system functions are treated as both values and used data and more advanced functions can be created by chaining several functions to provide a more advanced output from simple data. This results in a very deterministic result where the result will always be the same due to the input being the same which will result in the same output (Szabó, 2017).

Redux uses these pure functions to handle states and an example can be seen in Figure 3.18, this means that they take the current state and an action and produces a new state in the return.

```javascript
1 // stay immutable and always return a new object
2 function addAuthor(state, action) {
3     return [...state.authors, action.author ];
4 }
```

Figure 3.18: A pure function example for Redux Wieruch (2017b).

In comparison MobX uses influences from object-oriented programming and reactive programming where instead the states are wrapped as observables and the data can be modified by plain setters and getter functions. This however makes MobX states mutable because they modify the state itself rather than create a new copy like Redux. This is more similar to the regular writing style of JavaScript and Python due to them natively supporting object-oriented styles (Python, n.d.; Hahnekamp, 2018). Because MobX is object-oriented and both TeCST and FriendlyReader are developed as object-oriented projects currently time can be saved in implementation because they won’t have to be rewritten to a functional paradigm.
3.8.3 Logic and data encapsulation

Because of the object-oriented nature of MobX the data and logic are naturally encapsulated by the classes because related functions are placed in the same class as the data it modifies. While Redux does not have the same design pattern due to its functional programming nature, while they are related the states, reducers and actions live independently of each other. Figure 3.19 below illustrates a Comparison of user store in MobX and Redux where both examples achieve the same thing.

MobX forces the developer to logically encapsulate but that is the nature of object-oriented programming but provides a natural modularity in how the classes are developed. In comparison Redux does not require the developer to encapsulate the code in any specific structure which would by extension put all code structuring on the developer (Wieruch, 2017b).
3.8.4 Single or multiple stores

Redux is designed to have a single global store/state and that state is the single source of data with reducers being able to alter the immutable state. Technically Redux supports multiple stores since you can define multiple stores but it is not an advertised use of Redux and it
goes against what is suggested by Redux documentation (Redux FAQ: Store Setup, 2018). This centralized approach has one large benefit in that it simplifies undo/redo logic for state transitions and becomes very easy because the history of state transitions.

MobX by design uses multiple stores because each variable in itself is a state. But mostly MobX uses domain specific states through classes, by design the domain states themselves are not linked to each other in any way. This means that it allows multiple layers of states which might differ in importance but still want to be encompassed by the MobX logic you can keep these separate (Concepts Principles, n.d.). This however makes undo/redo state tracking more difficult because you need to create a global store manually that tracks the domain specific states, but gains the ability to choose which states should be tracked because it is a bespoke design.

### 3.8.5 Boilerplate

Redux is a very simple library with a short and easy to understand documentation which means that getting started with Redux is easy, however the design philosophy of Redux requires the developer to learn functional programming to be able to effectively use (Mobx vs Redux, n.d. Wieruch, 2017b) the library. Redux becomes exponentially more complex the more it is used (Kasireddy, 2017) and requires a stronger understanding of functional programming (Mobx vs Redux, n.d. Wieruch, 2017b).

With MobX developers will be able to use their experience if they’ve previously worked with popular languages such as C++, Java or Python (Wodehouse, n.d.) due to MobX using an object-oriented paradigm. This translates to a lower threshold to be able to use MobX to its full potential and makes MobX suitable for beginners (Wieruch, 2017b).

#### 3.8.5.1 Library Magic

A reoccurring word used when comparing MobX and Redux is the amount of magic the libraries provide, what is meant by magic in this context is the amount of internal implementation and logic the library has (Wieruch, 2017b). Redux is a very simple library and the main magic Redux provides is the store where each state transition is tracked and stored for future use but to modify the state everything has explicitly stated through reducers and actions this means that the developer has to create each part of the state transition.

MobX however provides several annotations of data and functions such as observable and action. These annotations abstract away a lot of logic which enables advanced implementations to be done in just a few lines (Wieruch, 2017b). The example in Figure 3.19 shows this in action where the same functionality can be made in fewer lines and provides a much easier to understand code even for beginners.

### 3.8.6 Deciding between Redux and MobX

Comparing Redux and MobX in practice boils down to comparing functional programming against object-oriented programming, both are valid and one could make the case that functional programming results in safer code because of the pure functions and immutability.

The problem with Redux is that the developer is required to learn functional programming which creates a huge threshold for everyone not previously familiar with functional programming. The library of Redux itself is easy to learn because the provided functions are very simple, but the power of Redux is in that it is written through functional programming. When the application grows it becomes more complex and due to the nature of Redux and functional programming it will become exponentially harder to work with and understand. It should be said that there is a clear trend towards functional programming and that it might become more common in the future (Wieruch, 2017b).

Take this in comparison with MobX where it uses an object-oriented approach seen in many popular languages such as C++, C#, Python, JavaScript, Java, etc. If a developer has
used any of these programming languages they will find similarities in MobX. Pairing this with the MobX annotations giving advanced behaviors with a single line of code makes it very suitable for beginners even if they don’t understand all the annotations (Wieruch, 2017b; Kasireddy, 2017).

Figure 3.20 by Preethi Kasireddy illustrates the learning curves of MobX and Redux. It shows the exponential growth for Redux with increasing complexity while MobX has a linear growth with increased complexity.

Figure 3.20: An illustration of the learning curves of MobX and Redux Kasireddy (2017)

Both of these libraries provide tools to implement all the required functional features identified in the breakdown of FriendlyReader and TeCST. To solve the problem both libraries are equally viable differing in only specifics on how the implementation is done and what separates them is the knowledge required from the end user (developer). Redux is a great library as it provides functionality that does not come naturally to MobX. One big drawback of Redux though is the aspect of functional programming. Comparing this to MobX, which uses object-oriented programming, MobX is friendlier for beginners and requires less intricate code to achieve similar results without the need to learn a new way of thinking if the end user has experience in object-oriented languages. MobX however requires a manual implementation of state transition history to enable undo / redo functionality to be able to go back to a previous state, which Redux provides automatically.
4 Implementation

With the issue of Redux requiring a knowledge in functional programing the obvious choice to implement state transition functionality in FriendlyReader and TeCST becomes MobX because of the similarity with the current implementation. However to provide the single store state transition history in MobX a new library called GlobalStore was implemented to fill this gap in MobX. This GlobalStore is a library that tracks classes with MobX attributes and tracks the changes done to the observable variables in every class in the substore which contains each tracked class. The entire library is compiled from NodeJS to a single JavaScript file and a source map that is used for debugging. The JavaScript file contains the entire GlobalStore library which can be seen in Figure 4.1 and is imported by the webpage through regular HTML.

4.1 GlobalStore library

In the Figure 4.1 the library is illustrated using a UML diagram and shows the interactions between the JavaScript files and how the data flows between the classes.

All external communication with the library goes through Database which has API functions that exposes the functions that the end user will use while at the same time hiding functions that are private.

GlobalStore is where the main logic is housed. It keeps track of all the substores and tracks changes in the child classes in the substores variable, stores are added and removed though the functions addStore(name, object) and removeStore(name). Using the function resetState() will make the last state the current state in the substore child.

UndoStore is where the states are stored in a list called snapshots this keeps track of the order the states were created and allows GlobalStore to pop and push states to the UndoStore.

Snapshots contains the actual logic on how to apply a state from the UndoStore to the substore child.
4.1. GlobalStore library

4.1.1 Database (filename: main.js)

Database contains the functions that are exposed to the client when importing the JavaScript file. All functions are prefixed with Database.function() for the client. The main function used by the client is initialize() which is a function called by the client at startup and it returns the actual GlobalStore class object to be used by the client.

DatabaseMobx contains a handle for the MobX library to be used for the MobX classes developed on the client. This removes the requirement of using a NodeJS environment to use MobX functions.

Database.API.SAPISRequest(...) is a request function to talk to SAPIS without having to use jQuery or XMLHttpRequest to communicate with the server. This is not strictly needed for the library to function but is a quality of life change for the library user. In a similar vein Database.utils.tokenizeText(...) is also a helper function to be able to tokenize text to convert a large text into a list of words. The helper function is later used in the example implementation.

4.1.2 Class GlobalStore (filename: store.js)

The class GlobalStore contains the logic for the library, the variable substores is observed by MobX (annotated by the @observable). This list contains all the external classes added by the function addStore(name, object) and removed by removeStore(name). Both annotated with @action due to them modifying an observable, name is the identifier for the class and is used as a key in the substores list i.e. substores[name] = object. In the GlobalStore constructor a MobX function called deepObserve recursively observes the observable objects (variables tagged with the MobX @observable annotation) in substores. The deepObserve reacts on an observable state change and pushes a state containing the afflicted data to the UndoStore class.

The last function resetState() is annotated as an @action because it modifies an observed state. The function is used when the client want to revert to the previous state, it does this by calling UndoStore.popSnapshot() and passing that state to the resetSnapshot(state, GlobalStore.substores) this returns a new absolute state which then overwrites the substores variable.

4.1.3 Class UndoStore (filename: UndoStore.js)

The UndoStore class contains all the states in the snapshots variable. One notable thing is that it is not an observable but instead is an untracked list while still being part of the MobX struc-
4.2. Synonym suggestion implementation example

ture. pushSnapshot(snapshot) pushes a new state into the snapshots variable while popSnapshot() removes and returns the previous state. lastSnapshot() returns the previous snapshot without removing it from the snapshots variable.

4.1.4 Snapshot.js

The Snapshot.js file contains the function resetSnapshot(snapshot, state). It takes the snapshot/state you want to revert and the state in which to apply that state change on.

4.2 Synonym suggestion implementation example

To explain how GlobalStore library is used in a real world example we use the Synonym feature of both FriendlyReader and TeCST. The two classes SynonymStore and TextStore are two classes written by the web developer and both of them use MobX tagging to define actions, observables, etc. The library then tracks these variables to provide a state transition logging for the user interface which uses the GlobalStore to input and receive data from the underlying classes. This is just an example of what the store is capable of, there is no defined limit on how many instances or classes it can track. The Figure 4.2 illustrates the way data flows between the UI, GlobalStore and its underlying classes to achieve this result. This is how I chose to implement synonym suggestion feature but that does not mean it is the only way to implement a synonym suggestion system.

The implementation was done with several features of FriendlyReader and TeCST in mind and even though they are not relevant to a basic usage example it provides a real example. This is one of the main reason why TextStore, which contains the original text, is an external class is to enable it to be used with other functions in the future such as replacing sentences or simplification which could be other classes in a similar vein as SynonymStore and TextStore.

![Figure 4.2: UML-diagram of Synonym implementation.](image)
4.2.1 User Interface

The user interface itself will always subscribe to the data within the GlobalStore to create a single state for the entire application which can be reverted. If the user interface would change something on the page without going through the GlobalStore it would create a desynchronized state. All the data used by the user interface and the data created by the user is read and stored in the GlobalStore.

The user interface look and logic is generated by the user interface GlobalStore and is the database/data storage bin for the underlying data and handles data processing logic.

Figure 4.3 shows the final example interface where the user interface is provided with synonyms through GlobalStore. When the users hovers over a yellow market word the top three synonyms are suggested and clicking a word replaces the original word. This is propagated to TextStore where the data is stored.

![Figure 4.3: Selecting a synonym and a dropdown with the top three choices for the Swedish word “vid”.](image)

When the user has replaced the word it gets replaced in the user interface with the word. In Figure 4.4 it shows how the word has been replaced, the user interface replaces the word and also reports this to TextStore.

![Figure 4.4: The synonym changed by the user to “samman”.](image)

4.2.2 TextStore

The TextStore class a simple class in terms of actual logic, it takes a long text string such as “This is a string” and turns it into a list of “This”, “ “, “is”,” “”, “a”, “string” though the function Database.utils.tokenizeText. The text string is then stored in this tokenized state in an observed variable called tokenizedText. The reason for this is that the application as a whole needs to be able to switch out individual words which we do with the SynonymStore. This provides state transition logging though the GlobalStore due to it being observed in the class, when a word is changed in the tokenizedText variable it is saved in GlobalStore.
Multiple words can be changed in the same go and only create one state by changing the value through a MobX action function. The main reason for this class is to enable the MobX and GlobalStore functionality for the text data object and create a central storage point for the text which encompasses all functions.

4.2.3 SynonymStore

The SynonymStore contains both the logic to retrieve synonyms but also the logic to request information from SAPIS. The class has an internal observed variable data which contains the response from SAPIS but also a few internal variables which handles the request metadata such as time, failed and requesting (if it is currently waiting for a response from SAPIS). The only data that _GS is watching is the data variable all the other variables can be changed without it being logged by GlobalStore.

analyzeText function sends a request to SAPIS, this function handles the request header options and the response message it receives. Input is a non-tokenized text and an optional callback function which is called when the response is sent back. The thinking behind the callback is the ability to execute functions as soon as they are available.

getSynonyms is a simple function to get the synonyms of a specific word in the context of the analyzed text ran through analyzeText. It takes the input of a word and the amount of synonyms and returns a list with the amount of synonyms sorted best to worst.

tagText function is the main function to produce the synonym change user interface for FriendlyReader and TeCST. The function takes a string text or a tokenized list and wraps each in a HTML <span> tag with a provided CSS class and what type of JavaScript event it should listen on, for example mouseenter. The function returns a tokenized list of all the words.

4.2.4 Code example to set up the GlobalStore for use

This chapter will illustrate how the library is used in practice using code.

The object Database is the library that contains GlobalStore which is the one exported through webpack and contains the API of the library. In Figure 4.3 _GS is initialized by Database.initialize() to activate a new Store object that tracks state transitions. This is where the main library logic lives. The classes SynonymStore and TextStore gets added into GlobalStore through _GS.addStore(key, classObject), these objects are now tracked by _GS and can be used through _GS.key.

The later part of the code just uses the different functions in the classes SynonymStore and TextStore to receive and modify data through _GS.Symstore.tagText, _GS.Symstore.analyzeText and _GS.TextStore.newText.
4.2. Synonym suggestion implementation example

```javascript
var _GS = null; // GlobalStore

$(document).ready(function () {
    // Initialize the store and keep track of it (or remove to reset)
    _GS = Database.initialize();

    // Creating the SynonymStore
    _GS.addStore("Symstore", new SynonymStore());

    // Creating the TextStore and add the text from the user (webpage element data)
    _GS.addStore("TextStore", new TextStore(textData));

    // Used to add a new text, but already set in the constructor above.
    // _GS.TextStore.newText(data);

    // Now we use the textstore to get the stored textdata.
    // This analyzes the text through a SAFIS request to get the synonyms for all the words
    _GS.Symstore.analyzeText(_GS.TextStore.text());

    // Takes a tokenized text and adds a <span> to each word and returns a tokenizedText with a span class
    // This class is used to give it the look the user interface designers wants.
    // _tagText(tokenizedText, listenEvent, callback, cssClass)
    let words = _GS.Symstore.tagText(_GS.TextStore.tokenizedText, "mouseenter",
        function(event) {
            // HTML data to create the look the user interface designers wants.
        },
        "synonyms"));

    // Join the words into a single text object by words.join(""")
    // Then add the data to the body to display the information.
    $('body').append(words.join(''))
});
```

Figure 4.5: Code stripped of all user interface code to illustrate the use of GlobalStore.
5 Discussion

This chapter will discuss the flow of choices made during the analysis and finally the implementation of the GlobalStore library.

5.1 Finding the right library

There is a large number of libraries on the market, each one trying to solve a different problem in a different way. Deciding on the actual criterions was the hardest part in this thesis, on one had it had to fit the size of the project at LiU, provide the necessary tools to complete basic state transition tracking and also have good documentation of its library. For example there are small libraries such as JavaScript State Machine that could have been used successfully in this project, it does however not come with a particularly good documentation but only a single page on a GitHub. This meant that the bigger libraries where the way to go in this thesis, looking around the web two choices became obvious Redux and MobX. Both libraries are extensively used on the web which proved that both have been used successfully and both had strong documentation with extensive examples explaining how to use the libraries. The differences between MobX and Redux are quite distinct with MobX being object-oriented and Redux being functional, there is a large difference between these two paradigms but they in the end accomplish the same result. Picking a library is a very subjective task in of itself because developers usually have a flavor on what they like and don’t like, such as different languages being their favorite. There is always a bit of subjective thought on what parameters are important, in this project the initial goal was to actually reduce unnecessary server traffic and state management, selecting for only these variables both Redux and MobX would work splendidly. But because the project was going to be used in the future we have to make it accessible to next developer. This is where these two libraries differ the most because they require different skills within programming itself. Redux uses as previously mentioned a functional paradigm while MobX uses an object-oriented approach. Normally people learn the object-oriented paradigm because most languages used for teaching are object-oriented e.g. Python, C++, C#, JavaScript, Java and so on. While functional programming is more an advanced technique used to avoid state-changing and mutable data which means writing the functions in such a way that it never accesses anything outside of the function neither read nor write. Instead all the information the function needs is inserted though parameters and any output data is returned at the end of the function. The idea behind functional programming is a simple one but can quickly become very hard to work with as we can see illustrated in Figure 3.20. A Redux library would have worked, but it is useless if the next developer cannot understand and expand the library. Another parameter to make the code usable is the way Redux and MobX defines their logic. Redux uses an explicit definition of each state transition, this requires all data that wants to be observed to be explicitly defined with actions and reducers on how the data should move. Whereas MobX instead uses a more implicit tagging system, it does not require the user to define how the data will change but instead tracks when the data change. MobX provides more implementation magic which means less code.
for similar function, which again is in favor of a less experienced developer where they don’t have to learn specific syntax for actions but instead just rely on a simple tag.

In essence the choice came down to which is easier for beginners, and finally the choice became MobX due to its ease of use for beginners specifically and a more familiar programming paradigm. It should be said that Redux might have been chosen if only experienced programmers where going to work on it, due to immutable object and pure functions it produces a much safer code because you always know a function will return the same value given the same input which in turn is more valuable even taking into account the more advanced codebase it would produce.

5.2 Putting it all together

The main idea behind the implementation and GlobalStore is breaking apart the problem into smaller pieces creating smaller storage areas that are then added to GlobalStore which then tracks the modules to provide state management. We can observe this in Figure 4.2 where by breaking apart the different modules we still get the functionality of the entire application but now we have a larger toolkit. Let us use SynonymStore as an example in the context of itself it is a dumb class as all the data required to function has to be sent in from an outside source. This module can be reused and applied to any situation where it would be useful. Creating the modules this way also allows new modules to be added without running the risk of breaking other modules.

FriendlyReader and TeCST share a large number of features the formation of the GlobalStore class storage system nudges developers to split up the modules into smaller ones because of how they want states to be tracked. But nothing unfortunately stops the developer from creating one giant class and putting everything in there.

The GlobalStore library is a simple library that only tracks observables which the developer chooses to add into the store using addStore and saves the states internally, internal data can be read and changed from the outside if one so desires but is not a normal use case. It then provides a resetSnapshot command that reverts to the previous state.

Lastly to efficiently handle server requests data in an efficient way is supported by GlobalStore but it also requires the implementation itself to support this and build a class that handles this through observables.

5.3 Problems and improvements

GlobalStore was finalized early on because its logic is simple, the change tracking logic to see if an object is different from before however was remade several times but was in the end replaced by a built in MobX function called deepObserve.

Currently redo functionality is not implemented but all the pieces are in place and it requires little work to accomplish.
6 Conclusion

The intended future developers of FriendlyReader and TeCST steered the decision of potential libraries to Redux and MobX but MobX was finally chosen due to its ease of use for beginners compared to Redux. Using MobX a global state management library was created called GlobalStore which handles the state transition logic to produce a single global state which enables the application to undo state transitions. GlobalStore improves the efficiency server requests by allowing the developer to save the data in states, it does however require implementation outside of GlobalStore. Two things where developed in code, a generic GlobalStore library that uses MobX to track data changes but also an implementation example for future reference to prove that the library works as intended but also to work as a documentation help for the future on how to use the library.
Bibliography


Bibliography


