# ASSIGNMENT OF BACHELOR’S THESIS

| | |
|---|---|
| **Title:** | A State Management in Multi-client Single Page Web Applications |
| **Student:** | Tomáš Bydžovský |
| **Supervisor:** | Ing. Marek Skotnica |
| **Study Programme:** | Informatics |
| **Study Branch:** | Web and Software Engineering |
| **Department:** | Department of Software Engineering |
| **Validity:** | Until the end of summer semester 2019/20 |

## Instructions

Modern web applications replaced most of the desktop software. This meant more requirements on dynamic communication with a server and quality of user experience. Users can also access an application from multiple clients, and they expect to have smooth experience.
A goal of this thesis is to compare state management approaches in such a scenario and demonstrate them on a google-drawings-like application.

- Review the state management techniques and approaches.
- Propose an architecture of a google-drawings-like designer using Redux and SignalR.
- Create a proof-of-concept open-source implementation of the proposed designer.

## References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 6, 2018

Bachelor's thesis

# A State Management in Multi-client Single Page Web Applications

***Tomáš Bydžovský***

Department of Software Engineering
Supervisor: Ing. Marek Skotnica

May 15, 2019

# Acknowledgements

I would like to thank my supervisor Ing. Marek Skotnica for his guidance and valuable advice.

I would also like to acknowledge my colleagues, who also worked on the project, of which this thesis is part.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 15, 2019 . . . . . . . . . . . . . . . . . . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Tomáš Bydžovský. All rights reserved.
*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### Citation of this thesis

Bydžovský, Tomáš. *A State Management in Multi-client Single Page Web Applications*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

# Abstrakt

Tato práce se zabývá správou stavu webových Single Page Aplikací (SPA) a jeho synchronizací na více klientech.

V posledních letech rostoucí složitost webových aplikací vedla k vytvoření nových systémů a návrhových vzorů, které se zabývají správou stavu. Přesto nejsou koncepty správy stavu příliš známé a nabídka existujících řešení může být nepřehledná.

Tato práce popisuje koncepty správy stavu a synchronizace dat v reálném čase. Součástí práce je analýza současných řešení a technik, zejména návrhového vzoru Redux a jeho implementace v podobě knihovny NgRx a také protokolu WebSocket a jeho využití v knihovně SignalR. V této práci je navržen design systému pro správu stavu webové aplikace se synchronizací v reálném čase s využitím návrhového vzoru Redux a knihovny SignalR. Jedná se o aplikaci podobnou aplikaci Google Drawings. Aby bylo možné demonstrovat nové koncepty a možnosti takového systému, byla podle navrženého designu vytvořena ukázková implementace ve formě prototypu.

**Klíčová slova** vývoj webových aplikací, správa stavu, synchronizace v reálném čase, Redux, NgRx, Angular, SignalR, BaaS, TypeScript

# Abstract

The subject of this thesis is state management of web Single Page Applications (SPA) and its synchronization on multiple clients.

In recent years the increasing complexity of modern web applications lead to the creation of new systems and design patterns, that deals with state management. However, concepts of state management are not yet widely known, and a wide range of existing solutions can be confusing.

This thesis describes the concepts of state management and real-time data synchronization. The thesis contains a review of current solutions and techniques. The main focus is Redux design pattern with its implementation NgRx, as well as WebSocket protocol and its use in the SignalR library. In this thesis design of a state management system with real-time synchronization for web application similar to Google Drawings using Redux design pattern and SignalR library is proposed. To demonstrate new concepts and capability of such system proof of concept implementation based on the proposed design was created.

**Keywords** web applications development, state management, real-time synchronization, Redux, NgRx, Angular, SignalR, Baas, TypeScript

# Contents

**Introduction** **1**
Motivation . . . . . 1
Goal and the Objectives . . . . . 2
State of the Art . . . . . 2

**1 Current state of State Management** **3**
1.1 State Management Introduction . . . . . 3
1.2 State Management Patterns . . . . . 5
1.3 Real-Time Communication Techniques . . . . . 15
1.4 Review of the State Management Solutions . . . . . 19
1.5 Review of Real-Time Communication Solutions . . . . . 21
1.6 Review Conclusion . . . . . 22

**2 Design and Architecture** **23**
2.1 Design Limitations . . . . . 23
2.2 Client Architecture . . . . . 23
2.3 Server Architecture . . . . . 27

**3 Proof of Concept Implementation** **31**
3.1 Application Description . . . . . 31
3.2 Client-side . . . . . 31
3.3 Server-side . . . . . 36
3.4 Usage . . . . . 37
3.5 Testing . . . . . 37
3.6 Future Development Recommendation . . . . . 38

**Conclusion** **41**

**Bibliography** **43**

**A Acronyms** **49**

**B Contents of enclosed CD** **51**

# List of Figures


1.1 Web development growing complexity [11] . . . . . . . . . . . . . . 4
1.2 Rising popularity of state management [12] . . . . . . . . . . . . . 4
1.3 MVC design pattern [15] . . . . . . . . . . . . . . . . . . . . . . . . . 6
1.4 MVC design pattern with a complex data flow [18] . . . . . . . . . 7
1.5 MVC variations [17] . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.6 Flux design pattern [1] . . . . . . . . . . . . . . . . . . . . . . . . . . 9
1.7 Flux design pattern in detail [19] . . . . . . . . . . . . . . . . . . . . 10
1.8 Flux design pattern with a complex data flow [18] . . . . . . . . . 10
1.9 Redux design pattern [22] . . . . . . . . . . . . . . . . . . . . . . . . 11
1.10 Redux design pattern with middleware [22] . . . . . . . . . . . . . 12
1.11 Akita design pattern [6] . . . . . . . . . . . . . . . . . . . . . . . . . 13
1.12 MobX design pattern [27] . . . . . . . . . . . . . . . . . . . . . . . . 14
1.13 VueX design pattern [5] . . . . . . . . . . . . . . . . . . . . . . . . . 16
1.14 High level comparison of Short/Long polling and SSE [30] . . . . . 17
1.15 BaaS – architecture and essential components [34] . . . . . . . . . 19

2.1 UML package diagram of client-side . . . . . . . . . . . . . . . . . . 24
2.2 UML diagram of the data model [46] . . . . . . . . . . . . . . . . . . 25
2.3 UML component diagram of client-side . . . . . . . . . . . . . . . . 26
2.4 UML class diagram of client-side . . . . . . . . . . . . . . . . . . . . 28
2.5 UML sequence diagram of client-side state change caused by action dispatched from UI component . . . . . . . . . . . . . . . . . . . . . 29
2.6 UML sequence diagram of synchronization side effects caused by action dispatched from UI component . . . . . . . . . . . . . . . . . 29
2.7 UML sequence diagram of client-side state change caused by action received from server . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.8 Client-Server bidirectional communication . . . . . . . . . . . . . . 30
2.9 Servers role as action hub . . . . . . . . . . . . . . . . . . . . . . . . 30

3.1 Purpose of Angular services . . . . . . . . . . . . . . . . . . . . . . . 33

3.2 Communication diagram showing state change cycle . . . . . . . . 35
3.3 Redux DevTools console in action . . . . . . . . . . . . . . . . . . . . 37
3.4 Synchronization between two clients opened in separate windows, part 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.5 Synchronization between two clients opened in separate windows, part 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

# Introduction

## Motivation

Nowadays almost every web site is no longer just a collection of static pages containing only text, but thanks to new web technologies are fully interactive and filled with animations and other live content. Modern web sites can provide the same functionality as typical desktop applications, e.g., email client, spreadsheet editor, video player. The more an application can do, the more complex it is, the more data it needs. Therefore it is usually more challenging to manage all that data and keep them synchronized on multiple devices.

Conclusions of the thesis will provide enough information, for anyone interested, about how to choose a solution for their project. So mostly developers, especially front-end developers, are going to benefit from this. Besides them, users of their application can also benefit from it, because if the app uses some verified solution, it is less likely to be buggy.

State management as part of web software development is relatively new and may play a significant role in the future. There are very few comprehensive sources of this topic. Because of the reasons above and because of the increasing importance of state management I have chosen this topic.

In the first part, I will describe techniques of state management and state synchronization in detail. In the next part, I will design a solution to state management and synchronization of the web application. The last part deals with the implementation of the solution proposed in the previous part.

This thesis builds on the previous semestral project and is part of the larger project. The project's goal is to create an open-source web application that allows the user to generate smart contracts.

## Goal and the Objectives

A goal of this thesis is to review state management and real-time state synchronization approaches in modern web applications development and demonstrate them on a google-drawings-like application.

The first objective involves reviewing and comparing current state management techniques and approaches.

The objective of the second part is to propose an architecture of a google-drawings-like designer using Redux design pattern and SignalR library.

The objective of the final part is to create a proof-of-concept open-source implementation of the proposed designer.

## State of the Art

MVC is one of the oldest patterns that deal with the development of GUI applications and state management. It was proposed in the 1970s. Many applications still use MVCs variants.

Recently, new patterns and frameworks emerged. For example, the Flux [1] that was developed by Facebook, because MVC does not scale well to large application codebase. Flux is the application architecture used for building client-side web applications. In contrast to MVC, it uses a unidirectional data flow.

Another new pattern inspired by the Flux is Redux. Primary goals of Redux [2] are unidirectional data flow and predictability. It helps developers to write applications that behave consistently, run in different environments, and are easy to test. It also allows for time-travel debugging.

Moreover, there are other state management patterns. Meiosis is a simple state management design pattern based on the concept of streams. [3] MobX is functional reactive style system. [4] VueX and Akita heavily inspired by Redux pattern. [5, 6]

WebSockets [7] is a new protocol that enables two-way communication between client and server. Using the HTTP protocol, that was not possible before. The only solution was to use workarounds. Now we can use it to real-time data synchronization. There are several server/client-side libraries such as Socket.IO and SignalR that use WebSockets. [8, 9]

Backend-as-a-Service (BaaS), sometimes also called Mobile Backend-as-a-service (MBaaS) is a quite recent development in cloud computing. BaaS [10] provides developers with backend cloud storage and processing for their web and mobile applications. It also offers common features such as user management, push notifications and social networking integration.

Chapter 1

# Current state of State Management

This chapter deals with the theoretical side of the problem. It contains an introduction to the state management, definitions of current design patterns that tries to solve it, review of the solutions, which use those patterns, list of technologies, that can be used to enable real-time synchronization and review of their implementations.

## 1.1 State Management Introduction

Modern web sites evolved from simple web pages to fully functional programs – web applications. Web application could now process user inputs, change applications looks and composition and also communicate with server beyond basic HTTP requests. With this evolution, a new need to manage data, state of the application, has arisen.

In recent years many new web frameworks have been developed, and many of them adopted a new concept of a web application - Single Page Application. Single Page Applications use only one HTML page. Instead of requesting and loading the new page they only re-render current page based on users input, downloading new data and content only when needed.

These new web application composed of many components needed some system to manage states of the components, communicate changes between them and synchronize states across multiple clients.

The first option was obvious, to adapt patterns used in desktop application development, mainly those based on MVC. That worked for smaller application but proved to be difficult to scale in large applications like Facebook.

State management became an issue. There is an infamous incident involving bug in message notifications on Facebook, where Facebook would show unread messages notification, even though all messages would be marked as

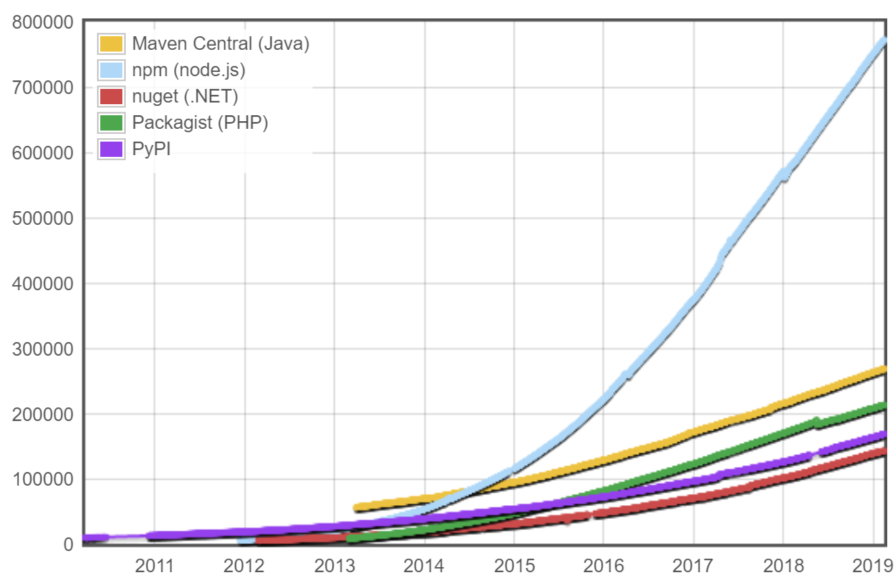
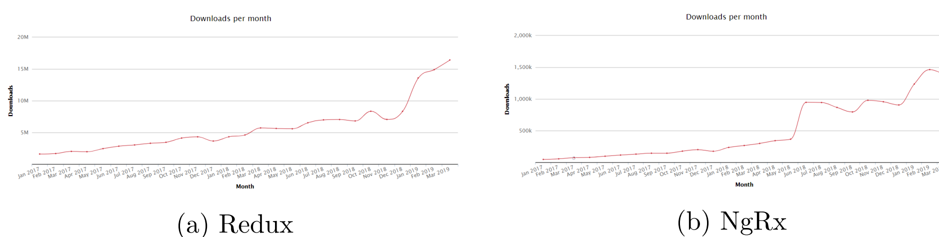



Figure 1.1: Web development growing complexity [11]

Number of usable modules in popular package managers from `www.modulecounts.com`.



(a) Redux (b) NgRx

Figure 1.2: Rising popularity of state management [12]

Number of downloads of ngrx and redux packages, two of the state management solutions on npm package manager.

read. Variation of this bug would appear again and again. That lead Facebook to focus on state management and create Flux pattern. Flux pattern is the first modern technique of state management.

Since then state management became a significant part of web application development, and many more solutions and techniques appeared.

Based on [13], there are so many options now that developers may find it difficult to choose the right solution for their project from overwhelming

choice.

### 1.1.1 Basic Definitions

State of an application is set of all the data in the application, not just a data model of an application. There are many types of state, server state, persistent state, URL and route state, client state, local UI state. Server state is stored on the server. Persistent state is part of server state, that is stored in some kind of persistent database. The URL and router state represents a loaded page and some parameters (e.g., index in a list of elements). Client state includes all data on the client. That can include data model, data not yet persisted to the server, server responses, cached data, and UI state. Local UI state is the state of an individual UI component, for example, the color of a text field checked checkboxes and so on. It is, of course, a subset of the client state. [14, 2]

State synchronization means keeping client state and server state synchronized, and in case of two or more clients using shared data, also keep all their state in sync. [14]

However, mutating state and using asynchronous actions simultaneously can be a problem. [2]

"As the requirements for JavaScript single-page applications have become increasingly complicated, our code must manage more state than ever before. Managing this ever-changing state is hard. If a model can update another model, then a view can update a model, which updates another model, and this, in turn, might cause another view to update. At some point, you no longer understand what happens in your app as you have lost control over the when, why, and how of its state. When a system is opaque and non-deterministic, it's hard to reproduce bugs or add new features." [2]

## 1.2 State Management Patterns

### 1.2.1 MVC pattern

Model-View-Controller and its variations like Model-View-Presenter and Model-View-ViewModel are well-known design patterns used for the development of applications with user interfaces for a long time. Their main goal is separate concerns of these parts. [15]

MVC has 3 main components:

- Model
- View
- Controller [16]

The model manages applications data. It holds the data (applications state) and business logic. An application can use more than one model, and the model can be dependent on other models. Change in one model triggers update in models dependent on this one. Model notifies views or controllers (it depends on implementation) subscribed to it about an update. The model provides an interface for updating model, that controller can use. [15, 17]

The view is composed of UI elements, which displays models data. Views are updated after state update in the model, which they are displaying. Views can be notified about the update by model, when using observer pattern, or directly by the controller. Applications usually have many views.

The controller handles user input, contains application logic, manipulates the model and can update the view. The controller receives input and process it. Based on the input it would update the model and invoke view update. There is a one-to-many relation between controller and view.

The main goal of MVC is a separation of concern, to separate presentation from model and view from the controller. That improves modularity, reusability, and testability of a system. [16]

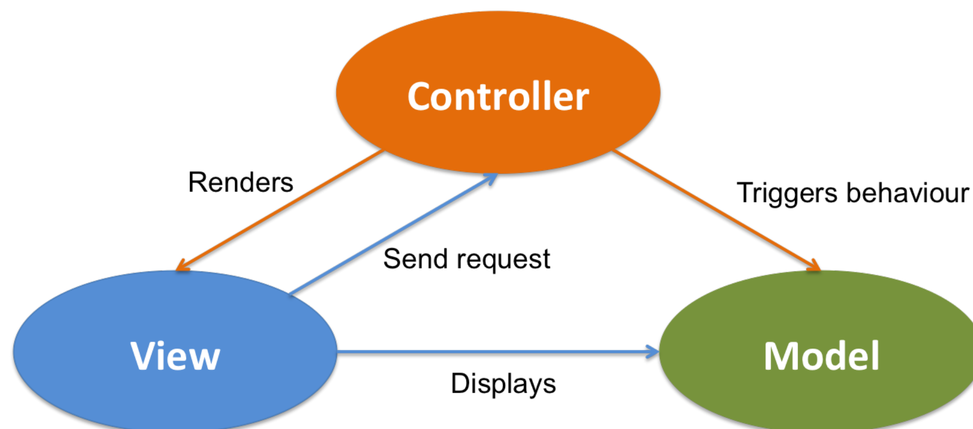



Figure 1.3: MVC design pattern [15]

MVC has few variations, most notably Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM). [16]

MVP has 3 main components:

- Model
- View
- Presenter

The model is the same as in the MVC. The view is also similar, but here it is handling user input and invoking action in the presenter. Presenter contains application logic, controls the view and manipulates with the model. There

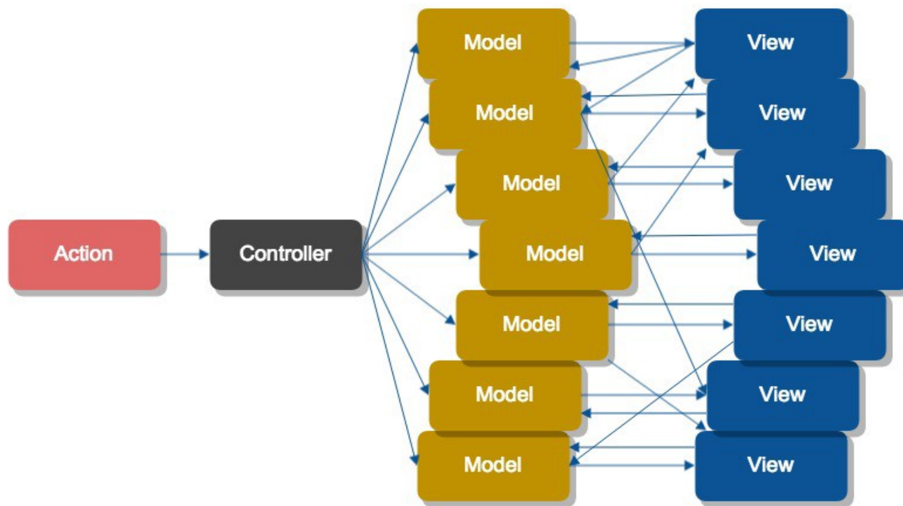



Figure 1.4: MVC design pattern with a complex data flow [18]

is one to one mapping between view and the presenter. The view can have a direct connection to the model, but there can also be a so-called passive view, that is not aware of the model, and all data transfer is controlled by the presenter. [16, 17]

MVVM has 3 main components:

- Model
- View
- ViewModel

The model is again the same as in the MVC. View handles user input and display data. The view is not directly aware of the model. ViewModel is a combination of Model and Presenter. It contains application logic as well as data. View and ViewModel are connected by bidirectional data binding. Many views can be bind to one ViewModel. [16, 17]

### 1.2.2 Flux

Flux is a new design pattern/application architecture. Flux was developed to improve the scalability of huge web-based applications. Flux is enforcing unidirectional data flow, which prevents bugs and also simplifies debugging and testing. [1, 18]

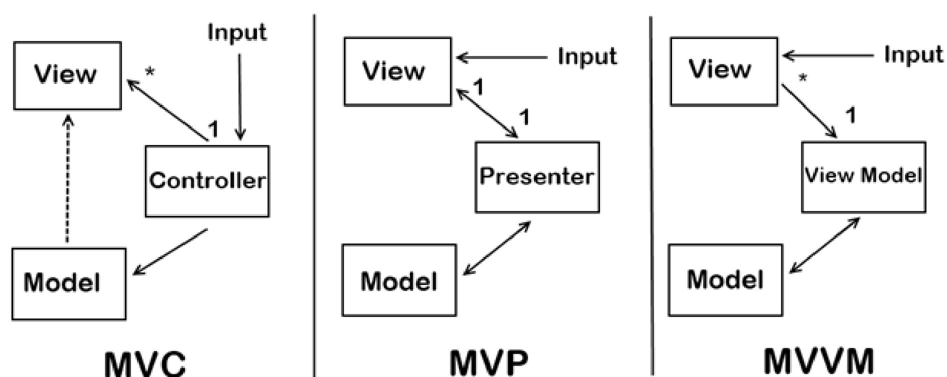



Figure 1.5: MVC variations [17]

In Flux there are 4 basic roles:

- Actions
- Dispatcher
- Stores
- Views [18]

Actions are simple objects, which contain information on how should the state be updated. Usually, actions consist of type and some payload. Type declares what should be done, for example: "ADD_NEW_USER" or "DELETE_ITEM." Type is a string value. The payload contains some additional information, like ID's, when it is needed. The payload can be any value or object. [1, 19]

```
Action = {
    type;       // mandatory parameter
    payload;    // optional parameter
}
```

Listing 1: Action definition

The dispatcher serves as a central hub for sending data. Stores can register its callbacks function with it. When the dispatcher receives actions, it processes them by invoking callbacks of all of the stores registered with it. When invoking a registered function, the dispatcher will pass the action to it, propagating information to the stores. There is an only single dispatcher. [1, 19]

Stores contain applications data and logic. The store is a source of truth. All views can draw data from the store. An application can use multiple

different stores simultaneously, every store managing piece of application state. Stores register their function callbacks with the dispatcher. That means that a single store can register more than one function and that more than one stores can react to single dispatched action. These callbacks functions are used to manipulate the state. The state can be either mutable or immutable. After changing the state, store notifies all its subscribed views about the update. [1, 19]

Views are implemented as controller-views. Views subscribe to stores and listen for the state change in the stores, to which they are subscribed. When notified by a store, the view gets a new state and re-render itself accordingly. Based on the user input, they dispatch new actions to the dispatcher. [1, 19]

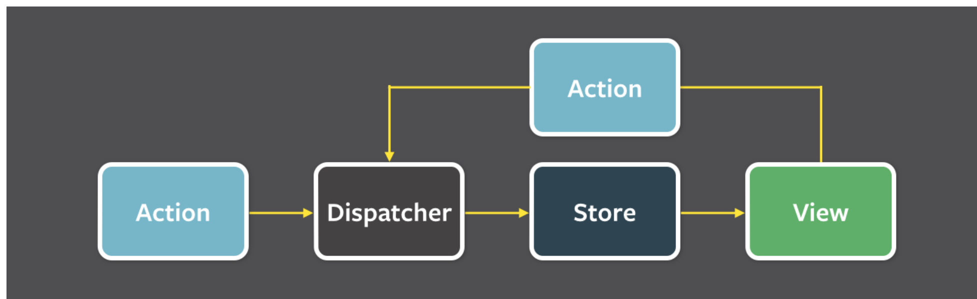



Figure 1.6: Flux design pattern [1]

Storing application state in store enables time travel debugging. That means that when the store is updating the state, it also saves the previous version of the state somewhere else. Then during the debugging, we can browse the history of all states.

Besides primary roles, there can also be Action Creators. Action Creators are helper functions, and they provide an abstraction for creating actions. Instead of creating actions manually in view, the view can call Action Creator, which returns created the corresponding action. [18]

Flux was designed with asynchronous calls in mind. Therefore Action Creators can make those calls and based on the response create appropriate action.

### 1.2.3 Redux

Redux is a design pattern/framework. Redux was heavily inspired by Flux, and they share many similarities, especially enforcing unidirectional data flow. That has the same advantages as in Flux. The main goal of Redux is to be predictable, which means that it behaves consistently, and is easily testable. [20, 2]

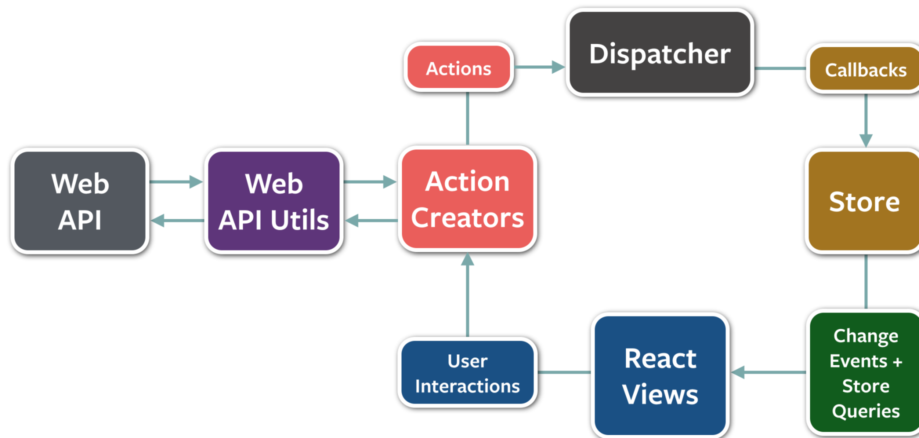



Figure 1.7: Flux design pattern in detail [19]

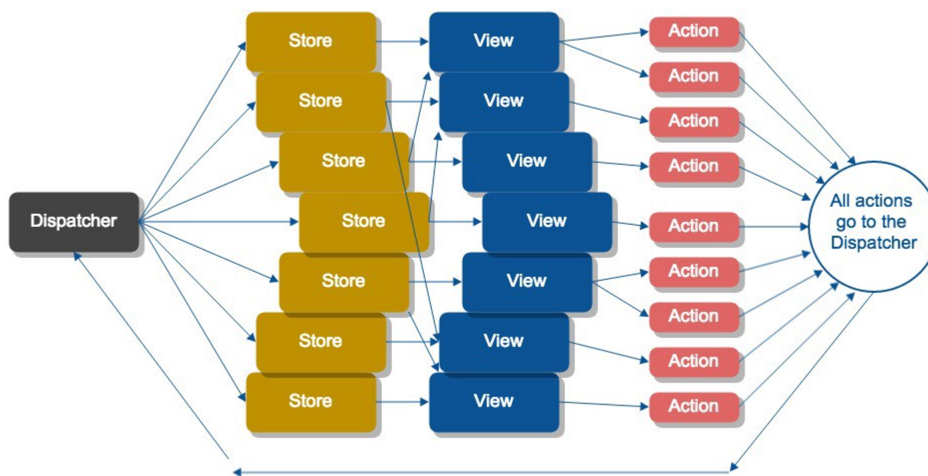



Figure 1.8: Flux design pattern with a complex data flow [18]

In Redux there are these basic roles:

- Actions
- Reducers
- Store
- views [21]

Actions are the same as in Flux. Reducers are pure functions. Pure functions are functions that use only input parameters. A pure function takes

inputs and based only on them outputs result value. That means that they do not have any side effects. Reducer in Redux takes two arguments, current state, and action, from which reducer creates a new state. [21]

```
function reducer (currentState, action) => newState
```

Listing 2: Reducer function definition

Reducers are passed to the store, during store creation. There can be multiple reducers, which is common practice, but in the end, they are usually combined into a single reducer. [21]

The store has a similar function as in Flux. It contains application state, but logic was moved to reducers. There can only be a single store, a single source of truth. It contains the state of the entire application. This global state is usually implemented as the state tree, states of all components arranged some tree structure. The state always has to be immutable. In Redux, the store receives action, which is then processed by the reducer. Reducer is being passed current state and action. New state generated by reducer is saved in the store and subscribed components are notified about the update. [21, 20]

View, same as in Flux, render itself accordingly to the current state in store and dispatches actions to the store.

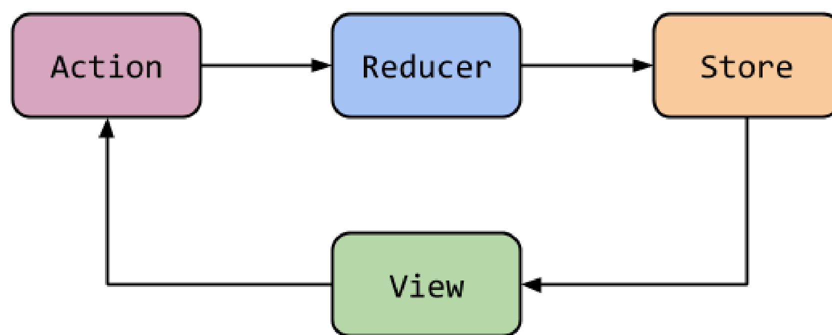



Figure 1.9: Redux design pattern [22]

Again as with the Flux, having state stored in the stores enables time travel debugging.

Besides primary roles, there can also be Action Creators. However, in this case, they are used solely for actions creation an initialization. That is because in Redux there is a concept of middleware.

Middleware is a function, that is applied to every action dispatched to the store. Because reducers are pure functions, they cannot have any side effect.

That is why Redux uses middleware, it does not change state, but it enables to action to have side effects. That is why in some implementations middleware is called side effect or just effects. Middleware is often applied in the store. [20, 2]

The store can have multiple middleware functions, each applied to every action. Middleware is used to make asynchronous API calls or to provide advanced logging options and so on. These side effects, for example, API calls, can result in another action being created based on results of the API calls.

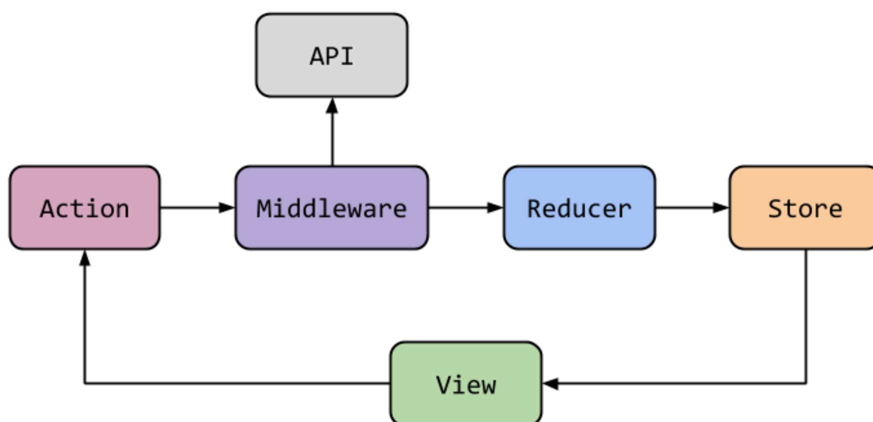



Figure 1.10: Redux design pattern with middleware [22]

### 1.2.4 Meiosis

Meiosis is a simple state management design pattern based on the concept of streams — stream of updates and stream of states. To manipulate with streams, it uses mainly JavaScript functions `map` and `scan`. [3]

### 1.2.5 Akita

Akita design pattern is based on Observables (Observable pattern). It draws inspiration from existing patterns, especially Flux and Redux. It takes the idea of multiple stores from Flux and the immutable updates from Redux and combines them with the idea of streaming data to create so-called Observable Data Store model. [6]

Akita is based on object-oriented design principles instead of functional programming typical for javascript applications. Its structure provides a fixed pattern that is hard to misuse. [23]

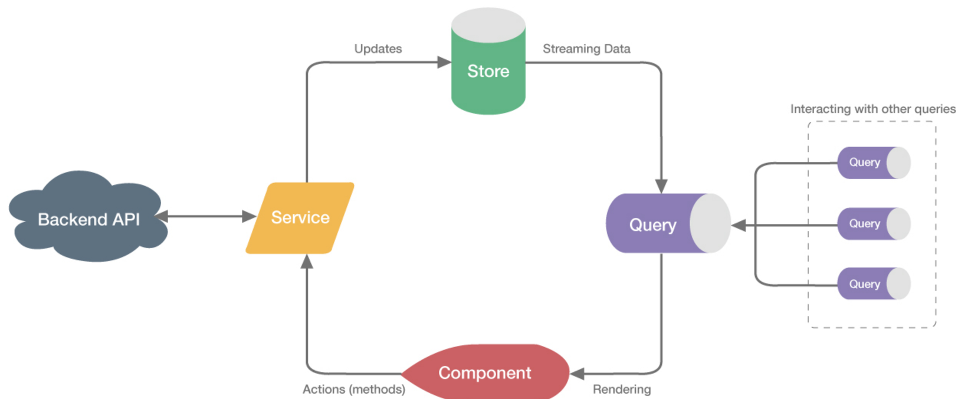



Figure 1.11: Akita design pattern [6]

Akita uses these main concepts:

- Model
- Store
- Query
- Service [24, 6]

The model represents data in the application. A store is an object which stores model. Store manipulates with the model using Data Manipulation Language methods (add, delete, update). The query is similar to a database query. Responsibility of a query is to query store and other queries. Query provides a variety of select methods to select a slice from the store. The query can return results as observables (reactive query), as well as simple objects (synchronize query). [6, 24]

Components should not interact with stores directly. They should use queries to get updates and use services to manipulate with the stores. Also, synchronous logic and server calls should be encapsulated in services and data services. [6, 23]

### 1.2.6 MobX

MobX is a simple, scalable state management solution, based on functional reactive programming. As with other patterns, MobX is heavily using Observables. MobX follows a simple, yet effective philosophy; "Anything that can

be derived from the application state, should be derived. Automatically." [4] That can include UI state, data serialization or server communication. That makes it impossible to produce an inconsistent state. [25, 4]

Many state management solutions try to restrict and control ways how the state can be modified, usually by making state immutable. However, that comes with its disadvantage: references to parts of a state can no longer work properly, which means, that it is harder to use data binding for example.[25] Core concept of the MobX:

- State
- Derivations (Computed values)
- Reactions
- Actions [26, 4]

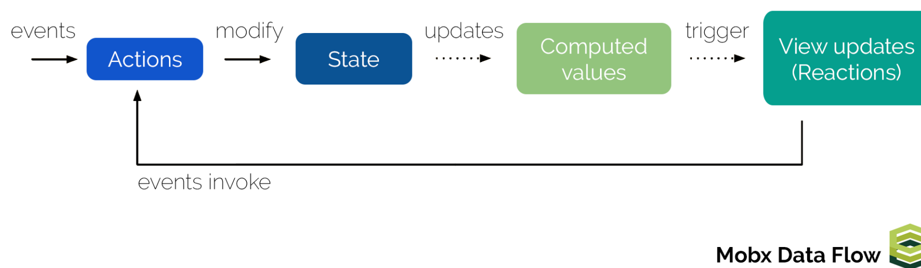



Figure 1.12: MobX design pattern [27]

The state contains data. Derivations are any data or values, that can be computed automatically from the current state. Derivations can be simple as one computed value (e.g., sum) or more complex, like rendered user interface, or serialized data. They are implemented as functions, that takes the state as input and returns computed values as output. Reactions are similar to derivations, but they do not produce any value. They are more like side effects. Derivations are performed automatically after state changes to perform some task. Usually, this is I/O related, like server calls. Actions are all the things that change the state. Actions and only actions should be able to change state. Every change of state is synchronously processed by the derivations and reactions. [25, 26]

### 1.2.7 VueX

VueX is state management pattern. It is inspired by the Flux and by the Redux. It uses a single store as a single source of truth. However, a state inside

the store is mutable. The store is reactive, which means it uses observables. [5, 28]

Core concept of the VueX:

- Getters
- State
- Mutations
- Actions [5]

Getters are like computed properties of the state. They are similar to derivations from MobX and Queries from Akita because they can get a slice of state or some data derived from the current state. The state is a state of the entire application and its stored in the store.

Mutations are ways of mutating state. The mutation is composed of string type and handler like events. A handler function is responsible for changing state. Handlers cannot be called directly. Handlers are registered in the store to their respective mutations. A handler is invoked by committing mutation to the store. Mutations must be synchronous.

Actions are similar to mutations, their handlers are registered in the store, but instead of mutating state, the execute asynchronous calls and then commits mutations to the store. Actions are dispatched from the components. [5, 28]

## 1.3 Real-Time Communication Techniques

### 1.3.1 AJAX

AJAX stands for Asynchronous JavaScript And XML. It is the use of the JavaScript and XMLHttpRequest object to communicate with servers. It can send and receive information in various formats, including JSON, XML, HTML, and text files. AJAX is asynchronous, which means it can communicate with the server and exchange data in the background without interfering with the user interface. After receiving data back, it updates the page without having to refresh it. That makes the application faster and more responsive to user actions. [29]

AJAX consist of a group of technologies, such as javascript, HTTP, CSS, XML, JSON, and HTML, working together. AJAX is generally used to get data from a server (usually by consuming REST API resources) and to push data to a server.

AJAX is a form of one-way connection, from client to server. It uses HTTP protocol, which is based on request-response schema, to send data. There are a few ways to abuse AJAX to emulate two-way connection.

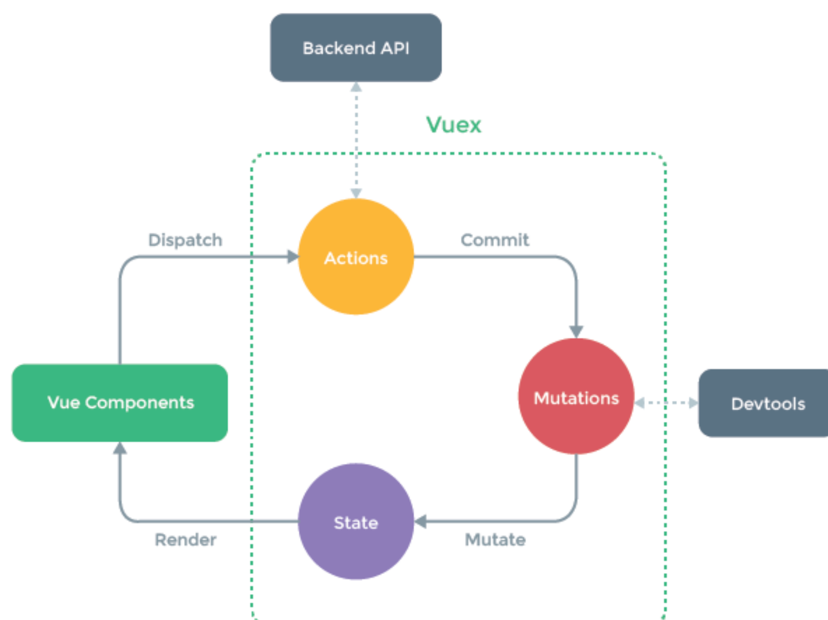



Figure 1.13: VueX design pattern [5]

One way is for a client to send a request to a server at regular intervals. That is called (short)polling or heartbeat. If the server has new data ready, it will send them. If there is no available data, the server sends a negative response. Another solution is called long pulling. A client makes request and server will hold the connection open until it can send data. [30, 31]

There is also a technology called Server Send Event, which is a form of one-way asynchronous connection from server to client. However, in reality, it is just long polling implemented in browsers to make it easier to use. [31]

### 1.3.2 Web Sockets

Generally, a socket is a communication channel connecting two endpoints. The WebSocket (WS) is an application layer communication protocol for full-duplex communication using TCP protocol. [31, 32]

The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that do not rely on opening multiple HTTP connections (e.g., using XMLHttpRequest and polling). [7]

Because HTTP was not meant to be used for bidirectional communica-

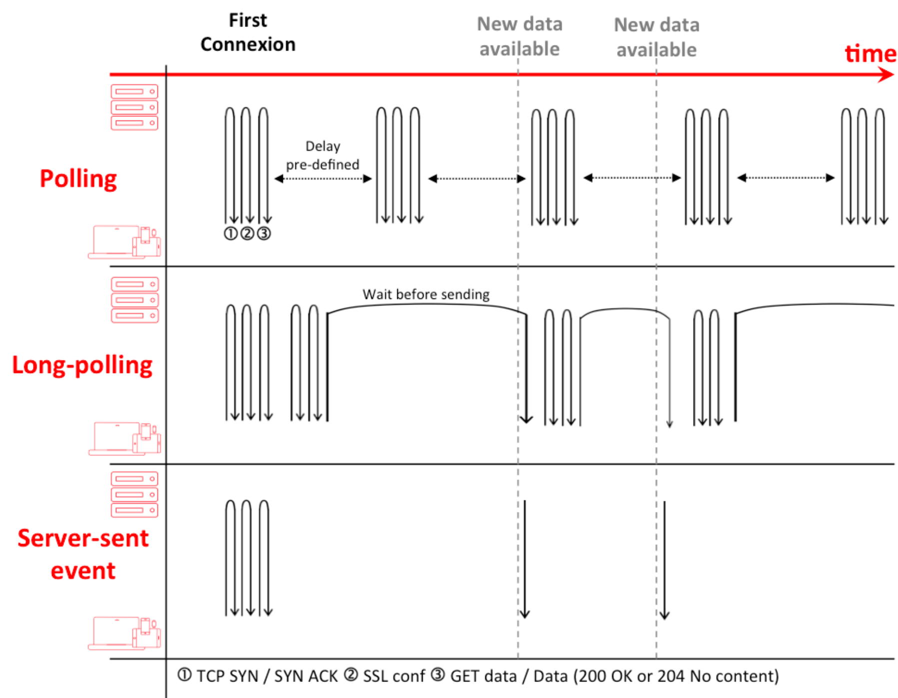



Figure 1.14: High level comparison of Short/Long polling and SSE [30]

tion, web applications that need two-way communication between a client and a server, had to abuse HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls. A simpler solution would be to use a single TCP connection for traffic in both directions. That is what the WebSocket Protocol provides. Combined with the WebSocket API, it provides an alternative to HTTP polling for two-way communication from a web page to a remote server. The same technique can be used for a variety of web applications, such as games, stock tickers, multiuser applications with simultaneous editing, user interfaces exposing server-side services in real time. [7]

The WebSocket Protocol is designed to replace existing bidirectional communication technologies that use HTTP as a transport layer and to benefit from existing infrastructure (proxies, filtering, authentication). The connection gets established by standard HTTP handshake. However, the protocol is designed to allow for a future change of the handshake mechanism. The WebSockets uses TCP port 80. The WebSocket also has secure version WebSocket over SSL (WSS) using TCP port 443. [31, 7]

### 1.3.3 Backend as a Service

Backend-as-a-Service (BaaS) is a type of cloud service. It sits somewhere between Platform as a Service and Software as a Service. BaaS allows developers to focus on the frontend of their application and to create backend functionality easily, without need to build it and maintain it. BaaS vendors provide pre-written software for usual server functionality. [10, 33, 34, 35] Common BaaS features

- User Authentication and Management
- Database and Data Management
- Server Scripts
- Push Notifications
- Messaging System
- Cloud Storage
- Real-time Synchronization
- Caching
- Offline and Client-server Synchronization
- Geolocation
- File and Media Management
- Analytic Tools [10, 33, 34, 36]

BaaS is composed of server part, set of APIs and optionally SDK. Server part provides application logic and database. Application logic can be implemented with server scripts. The database is generally some type on NoSQL, but it is not a rule. API can be REST API, Query style API or some other custom type. Purpose of the SDK is to make working with BaaS and its APIs more comfortable. [34, 33]

Using BaaS can have benefits, like faster development, development without the need for back-end developers, much better scalability, using already provided features (e.g., Real-time Synchronization) and easier communication with back-end when using SDK. [34, 33]

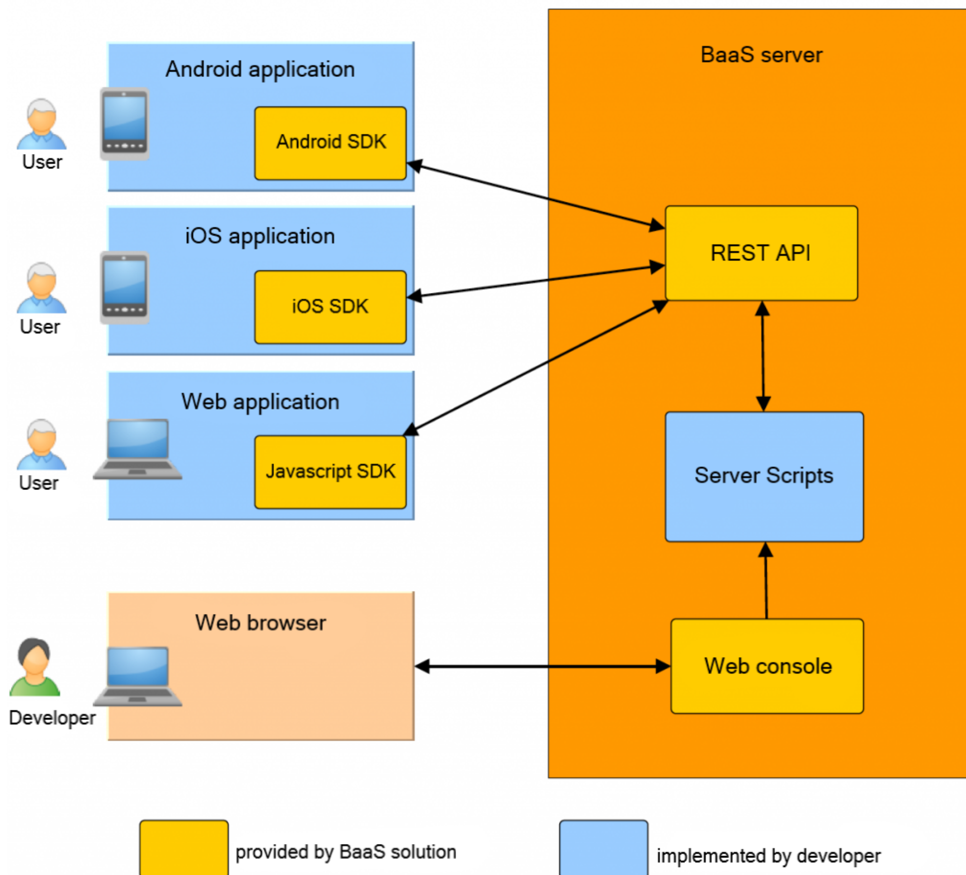



Figure 1.15: BaaS – architecture and essential components [34]

## 1.4 Review of the State Management Solutions

### 1.4.1 Flux

The Flux pattern has an only major implementation, the Flux library made by Facebook. Flux was the first new state management pattern and is still quite popular, but it has been overshadowed by other Redux like implementations. Flux, as well as other Flux implementations, are best suited to React framework. [1]

### 1.4.2 Redux

Redux library or Redux state management container is original Redux implementations. Redux is native to React, but there are bindings for other frameworks, making Redux very universal. In Redux store can be bound to a router of a SPA, changing state, based on the URL navigation. Redux also has a very powerful debugging tool called Redux DevTools, which provide

state changes logging, state visualization, time-travel debugging, actions logging and other options. Redux library is the most popular state management solution. [2, 20]

### 1.4.3 NgRx

NgRx library is a container for managing state, build on RxJS library (Observables) using Redux pattern. In NgRx middleware is called effects. NgRx can also use Redux DevTools. Same as Redux, the store can be bound to a router, which means the state can change navigation and navigation can change state. NgRx is part of Angular framework ecosystem, and it is the most popular state management solution for Angular applications. NgRx provides NgRx Entity adapter for managing record collections. The entity provides an API to manipulate and query entity collections. [37, 21]

### 1.4.4 NGXS

NGXS library is Redux-like state container. It can be used only by Angular application. It is newer and more modern than NgRx, because it uses more TypeScript and Angular features, like decorators and dependency injection. NGXS does not use reducers. Instead, it uses action handlers defined in the store identified by the decorators. Action handlers themselves can be asynchronous. To get slices of state NGXS uses selectors, like NgRx. This schema to change and read state is inspired by the CQRS pattern. NGXS store provides a snapshot method that creates a copy of the entire state. NGXS requires boilerplate code than Redux and NgRx. NGXS uses plugins to provide limited Redux DevTools support, save/load functionality for local and session storage, router binding. [38, 39, 37]

### 1.4.5 Meiosis

Meiosis provides simple light-weight state management system, with minimum boilerplate. It is meant for small applications. However, it is a small project, with lacking documentation and examples. [3]

### 1.4.6 Akita

Akita library tries to be simpler than Flux and Redux implementations. It has a moderate learning curve, and it reduces the amount of boilerplate code. Akita is based on OOP principle, rather than on functional programming. It was designed with Angular in mind (e.g., uses of services), but can work independently on the framework. Akita provides support for Redux DevTools (limited), router binding, storage persistence, and state snapshots. [37, 6]

### 1.4.7 VueX

VueX is state management library of Vue framework. It is designed with Vue ways of efficiently updating and rendering components in mind. VueX provides its own set of dev tools. [5]

### 1.4.8 MobX

MobX is not a state container like other solutions. It is more like a system of updates. It is based on functional reactive programming principle. Its goals are to be simple, scalable and reduce boilerplate code. MobX works best with React, but it can be used with other frameworks with the help of adapters. MobX provides developers with MobX DevTools. MobX is quite different from other solutions. Its creators often use spreadsheet analogy: the state is like a data cell, and computed values are like formulas and charts. [25, 4, 26]

### 1.4.9 Custom Redux implementation

There is always the option to create a custom implementation of some of these state management pattern with the help of RxJS. However, it is unnecessary, just like reinventing the wheel. It can be done in smaller projects, but otherwise, it can introduce more problems than it solves and using dev tools is hard or impossible. [37]

## 1.5 Review of Real-Time Communication Solutions

### 1.5.1 SignalR

SignalR is a library that provides an abstraction for two-way communication between server and client. It consists of two parts. One part is server-side ASP.NET or ASP.NET Core library used on the back-end. The other part is the client-side library. Client-side libraries are provided in several languages, eg. Javascript/Typescript, .NET or Java. [9]

SignalR enables real-time data transfer. SignalR by default uses WebSockets as the transport layer, but when that is not possible, it can also use other options, such as ”long-polling.” SignalR provides API for remote procedure call (RPC) to invoke functions on the client as well as methods on the server. [40, 41]

### 1.5.2 Socket.IO

Socket.IO is a library for bidirectional real-time communication between server and clients. It consists of Node.js server and JavaScript client library. The community provides client implementations in other languages, such as Java,

.NET, Python, C++, Swift and Dart. Socket.IO is similar to SignalR. It also uses WebSockets and long-polling as a fallback. [8]

### 1.5.3 Custom Web Sockets solution

Using just plain WebSockets can also be a solution, and even some of the state management libraries provide direct support. However, that means no fallback option, no abstraction over implementation and no utilities and extra features.

### 1.5.4 Google Firebase

Firebase is a BaaS solution from Google. It provides many features and tools. Real-time synchronization is the main focus of Firebase. Firebase has two types of real-time NoSQL databases Realtime Database and Cloud Firestore. Realtime Database is a JSON document storing data. Newer Cloud Firestore is based on documents and collections of documents, where document is a set of key-value pairs. Firebase is a truly serverless solution. It can be integrated with other Google services and tools. Firebase is one of the most popular BaaS solutions. [42, 43]

### 1.5.5 Backendless

Backendless is another BaaS solution. It uses MySQL database and all features that relational databases have, like powerful queries, relations, option to use ORM. Backendless provides the option to create custom REST-style APIs. Pro version could be self-hosted. Server scripts can be created by visual programming. Since version 5 Backendless provides a real-time database, which can be used to update data on clients. [44, 45]

## 1.6 Review Conclusion

In this chapter basic concepts of state management were explained, current state management patterns and real-time synchronization techniques were described, existing solutions using those patterns and techniques were reviewed. Information from this chapter is used in the next chapters.

Chapter 2

# Design and Architecture

This chapter deals with designing the state management system for web-based SPA. It contains a description of used technologies, client-side architecture and server-side architecture, both using techniques and solutions described in the previous chapter.

## 2.1 Design Limitations

Because this thesis is part of a larger project with its specification [46], there are some limitations. The most significant limitation is that client-side application is using TypeScript-based Angular web framework. So client-side code is written in TypeScript, using Angular framework. There are more Redux pattern implementations. For example NgRx, NGXS, and Redux. Because it is beneficial to use libraries targeted to your framework, I am using NgRx, which is the most popular Redux implementation in Angular ecosystem. When using SignalR, there are two possibilities, ASP.NET or ASP.NET Core. Because .NET Core is newer, scalable, cross-platform and because the .NET framework does not provide any significant benefits for this application, I have chosen ASP.NET Core version and corresponding client-side version.

## 2.2 Client Architecture

Client-side is part of Angular project. State management and synchronization part consist of these elements:

- Signalr service
- State management service
- Model module

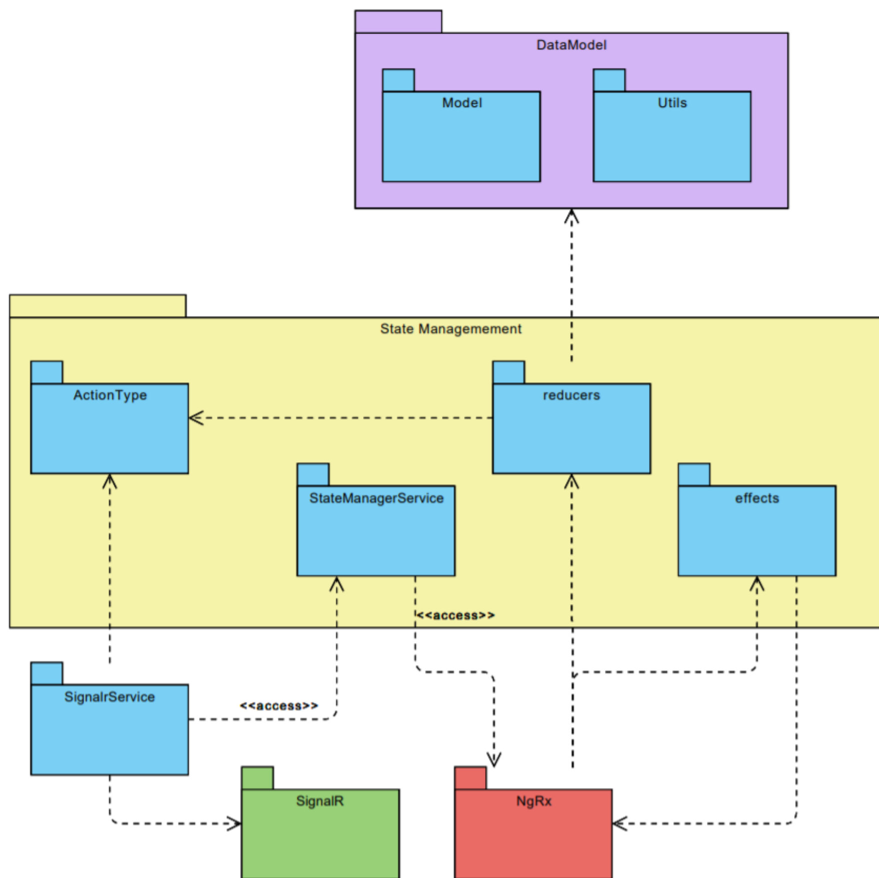



Figure 2.1: UML package diagram of client-side

### 2.2.1 Model module

The model represents the data layer of the application. The model consists of data and business logic. Model is the central part of the application state.

The module is divided into two parts. First one contains definitions of all the interfaces and the classes of the model, the other contains business logic in the form of functions, used to manipulate with the model.

### 2.2.2 State Management Service

State management service provides an abstraction for state management implementation in the application. Because we use Ngrx, the service communicates with the Ngrx store.

There are two main functions of this service. First is to receive actions from components and dispatch them to the store. Second is to allow components

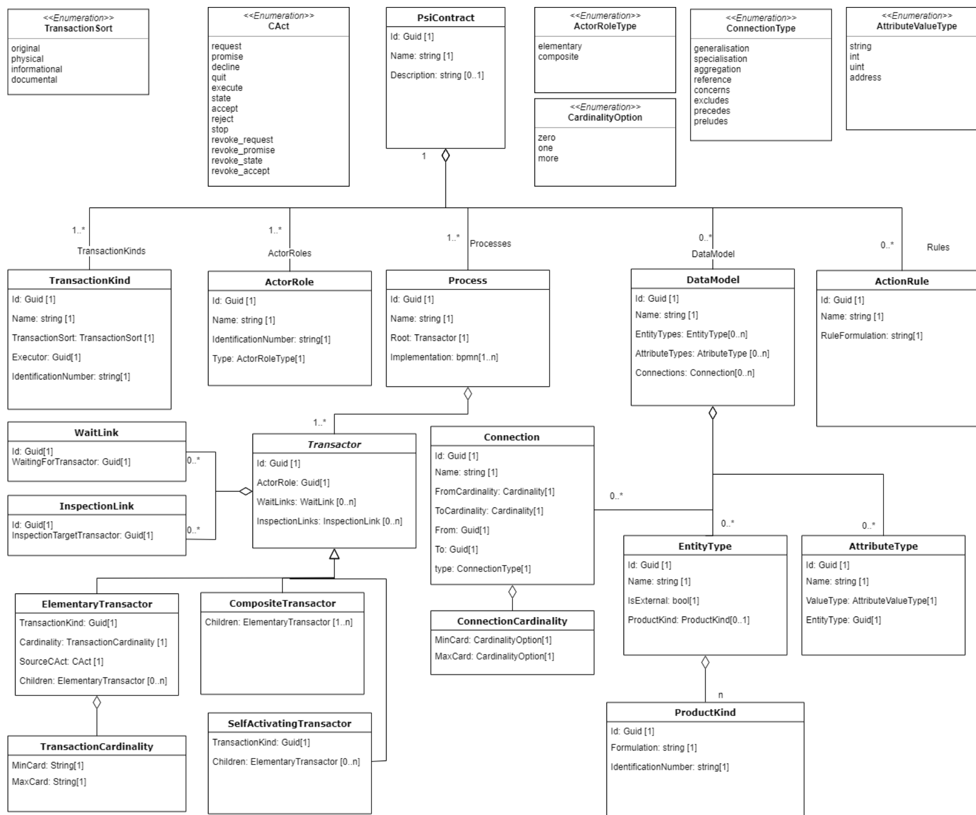



Figure 2.2: UML diagram of the data model [46]

to subscribe to the state changes. So it serves sort like a mediator between store and rest of the application.

Apart from that state management service also contains other crucial components of Redux: reducers, effects, action and action creators. Actions are implemented as instances of a class, not as plain objects, and constructors have the role of action creators. Action types are defined in the enumerable object. Actions have to implement `IAction` interface, that enforces action to have mandatory string type and an optional payload. Payload has to implement `IActionPayload` interface to ensure that any action that changes state has its identification. Declaration of these interfaces can be seen in the listing 3.

Reducer is implemented not as function, but as a static method of `PsiContractEditorReducer` class because Angular and NgRx allow it and its cleaner, more object-oriented design.

Side effects are inside the `PsiContractEditorEffects` class. The primary side effect is of course synchronization.

Application state, as defined in `PsiContractEditorState` class, is divided into three distinct parts: psiContract, undoStack, redoStack. That provides

```
export interface IActionPayload {
  actionId: Guid;
}

export interface IAction extends Action {
  type: string;
  payload?: IActionPayload;
}
```

Listing 3: Interfaces defining action and action payload

History of Changes feature, that enables undo and redo functionality. That means that a user can travel through the previous version of a model, like in word processor or web browser. Property psiContract is the current state of a model, undoStack contains past states and if the user has undone a few changes redoStack contains those undone changes.

Every psiContract has versionId property. It gets updated with every state change. New versionId is actionId of the action that caused this change.

### 2.2.3 SignalR Service

This purpose of this service is to deal with communication with the server. It is sending actions to server and is receiving actions from the server back. The state management service uses this service. It is a crucial part of real-time synchronization.

SignalrService is responsible for creating, configuring and establishing the connection to the server. It is also responsible for reestablishing connection in case it was lost. The API of this service provides the method to send actions and to get state of the connection.

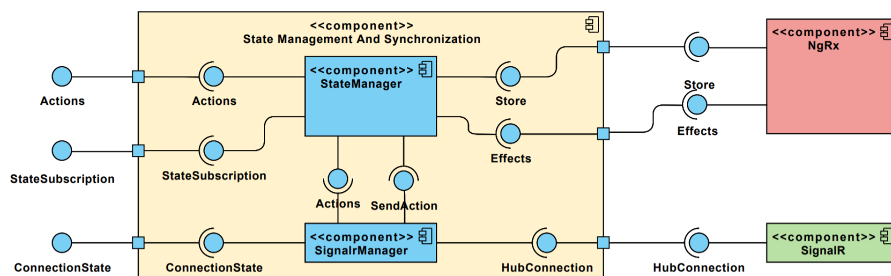



Figure 2.3: UML component diagram of client-side

## 2.3 Server Architecture

Server-side is very simple ASP.NET Core server application which provides web API to which clients can connect.

### 2.3.1 SignalR communication hub

The SignalR hub has a very significant role. Hub uses SignalR to communicate with clients. Hub defines a set of methods, that can be invoked from clients, or that can invoke a function on clients.

In the current situation, the server only resends action from one client to others, but in the future, it can be upgraded with to have its own Redux implementation and database persistence.

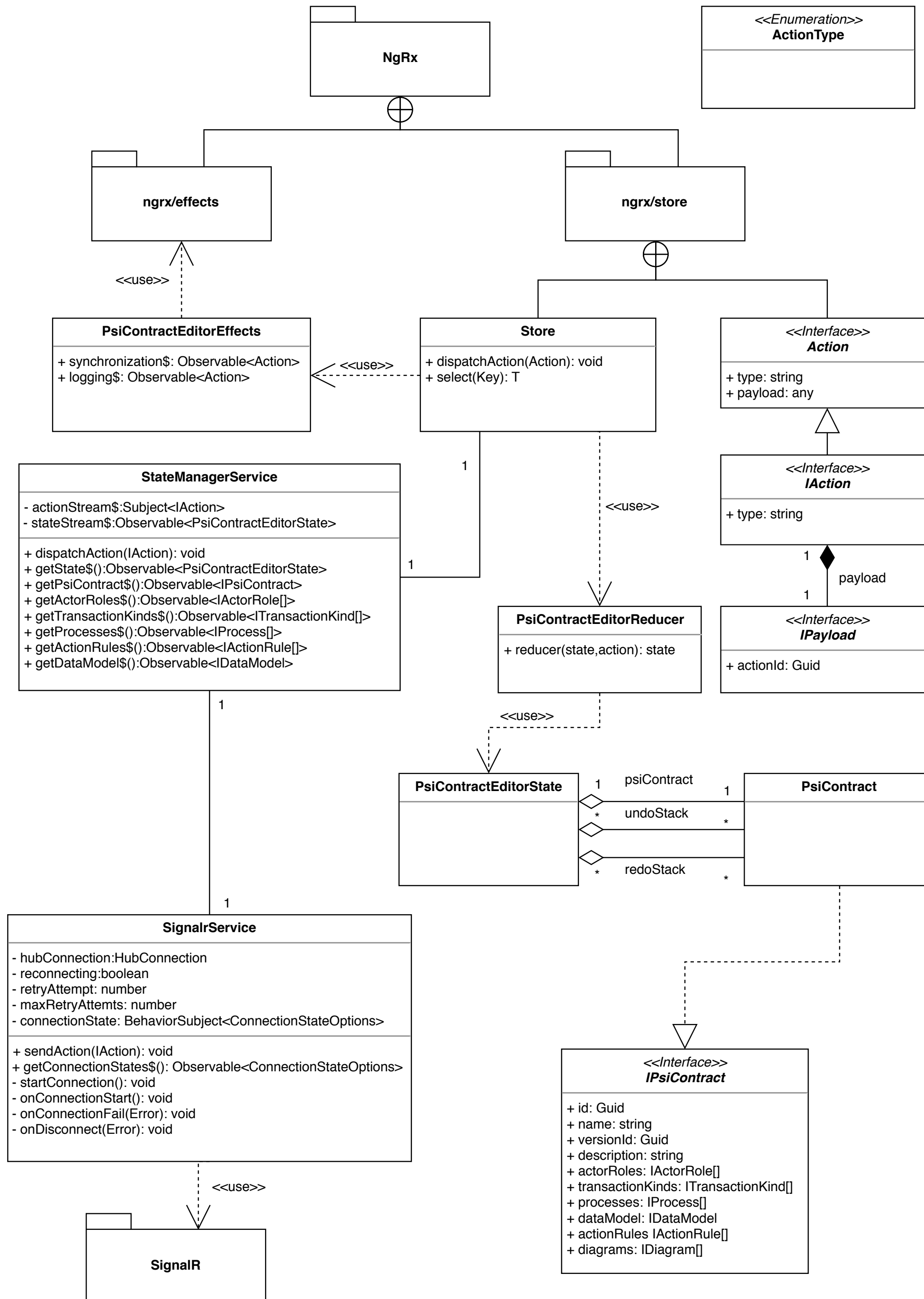



Figure 2.4: UML class diagram of client-side

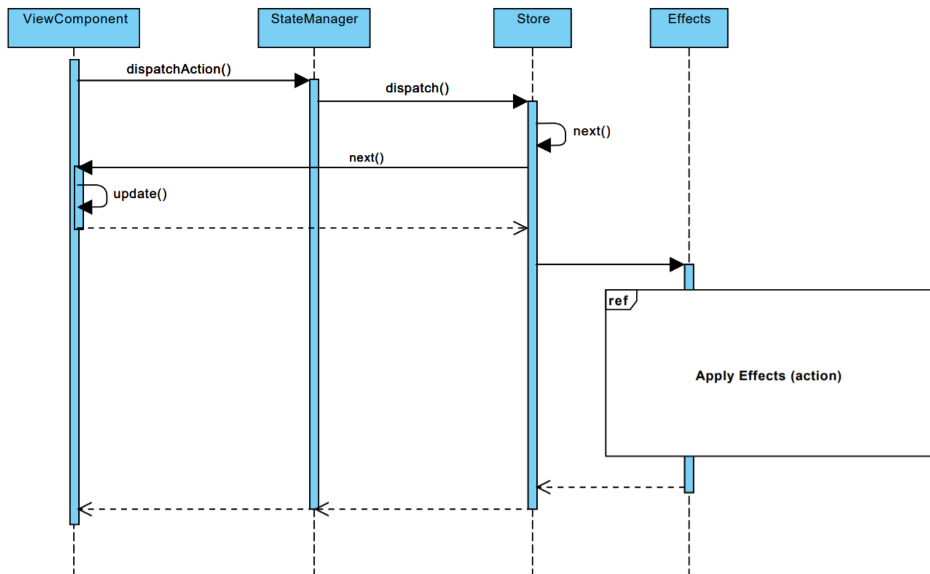



Figure 2.5: UML sequence diagram of client-side state change caused by action dispatched from UI component

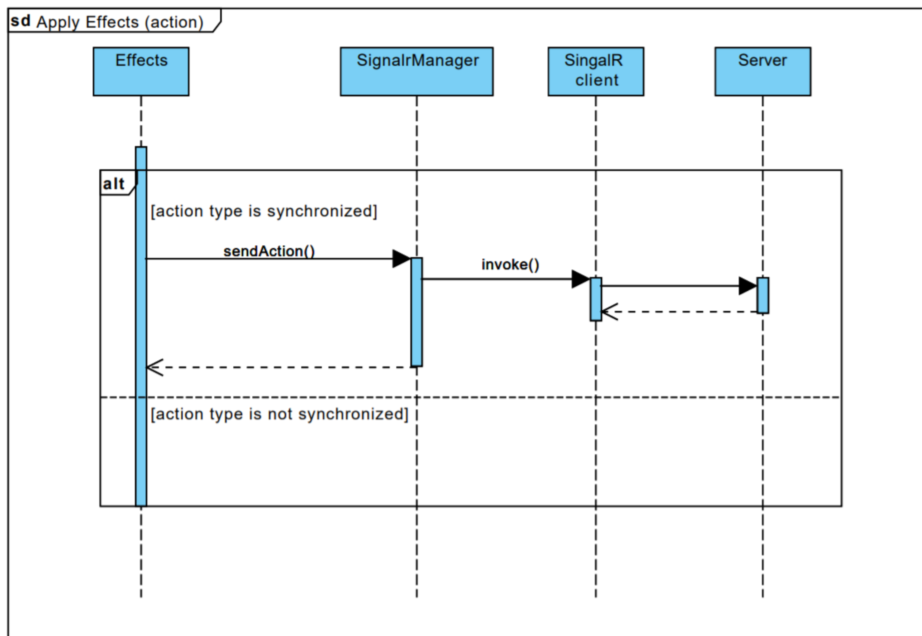



Figure 2.6: UML sequence diagram of synchronization side effects caused by action dispatched from UI component

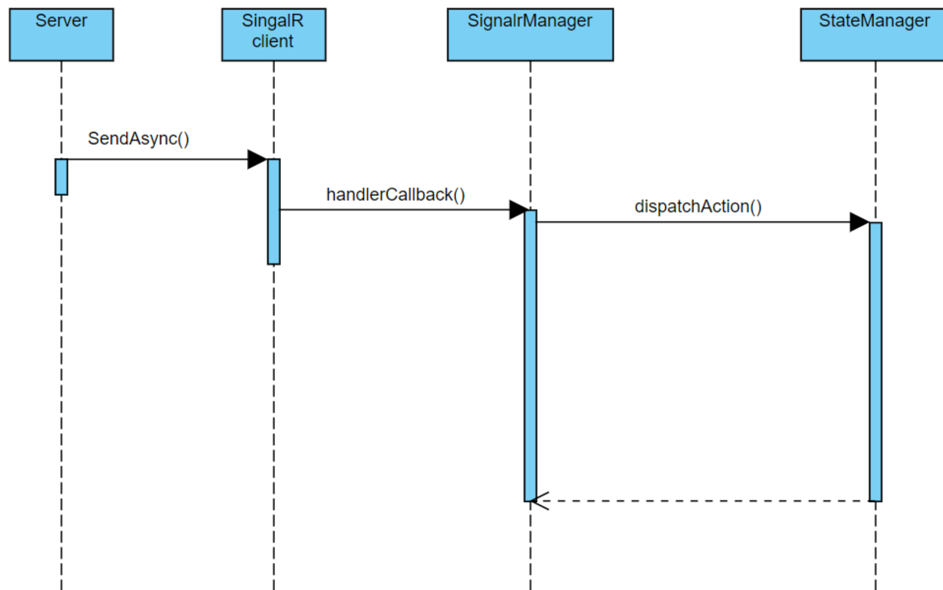



Figure 2.7: UML sequence diagram of client-side state change caused by action received from server

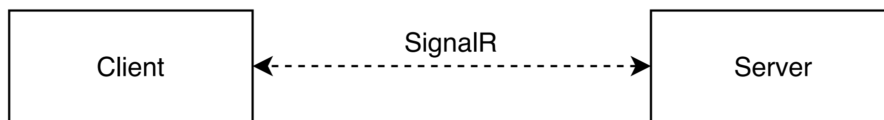



Figure 2.8: Client-Server bidirectional communication

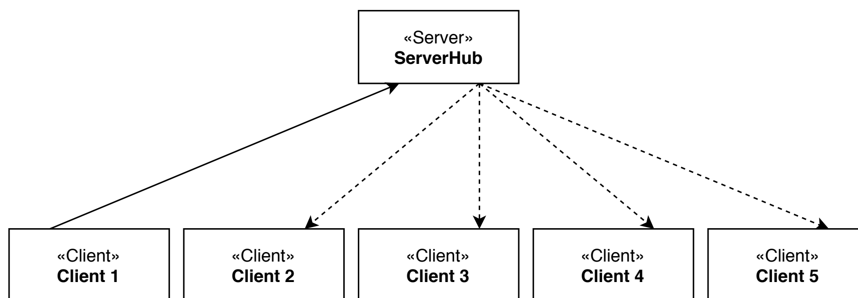



Figure 2.9: Servers role as action hub

Chapter 3

# Proof of Concept Implementation

This chapter deals with implementation details of proof of concept, based on the design proposed in the previous chapter. This chapter contains a description of the client-side part, description of the server-side part and description of the usage of this system from the perspective of a developer of the application and the perspective of a user of the application.

## 3.1 Application Description

The proof-of-concept of the state management system, which is this thesis about, is part of a web application called EnterpriseDesigner. It is a web application that provides online tools for process modeling. It consists of a set of visual editors and designers, that enable creating BPMN and DEMO, methodology models. The application is divided into two modules, client-side and server-side.

## 3.2 Client-side

The client-side part of the application is single page web application build using Angular framework, TypeScript, and other graphical technologies.

The application uses NgRx library to manage state. NgRx module is divided into @ngrx/store and @ngrx/effect packages. SignalR client is provided by @aspnet/signalr package.

- TypeScript version: 3.2.2
- Angular version: 7.2.0
- @ngrx/store version: 7.1.0

- @ngrx/effects version: 7.4.0
- @aspnet/signalr version: 1.1.2

### 3.2.1 Used Technologies

TypeScript language is a superset of JavaScript. It was created by Microsoft and released as open-source in 2012. Since then it was adopted by many large companies, for example, Google and Adobe. The main feature of TypeScript is type safety, which is missing in regular JavaScript. That provides TypeScript with IntelliSense, easy refactoring and syntax error highlighting. That allows developers to write a larger application because the code has become more manageable. [47]

In order to use TypeScript, the transpiler is needed. A transpiler is a special compiler that compiles code from one language to another language. The TypeScript compiler compiles TypeScript code to javascript code, that can be run in browsers or Node.js. [47]

TypeScript provides modern JavaScript features, such as classes, inheritance, and modules, but on top of that also static type checking, interfaces, and generics. [47]

The angular framework is a modern JavaScript framework for creating single page applications. It uses TypeScript. Angular is a product of Google. [48]

Angular and NgRx both use the RxJS library extensively. RxJS (Reactive extensions for JavaScript) provide Observable implementation. Observable pattern, sometimes called Subscriber pattern is a system where are two roles: Observable and Subscriber. Observable holds some data or value, that changes over time. We can look at Observable as an asynchronous stream of values. Subscribers, sometimes called Observers can subscribe to the Observable. When value inside Observable changes, the Observable will notify all its subscribers with a new value. Subscribers have to provide a handler, which will be executed when the subscriber receives a notification with a new value. Apart from that RxJS provides operators. Operators are a function which can be applied to the observables in order to manipulate with the data flow. For example to filter or change values or to provide time-related functions as debounce or delay. [21]

SignalrManager and StateManager are implemented as Angular services.

Services are special classes annotated with `@Injectable()` decorator. That marks them for Angular dependency injection system. The angular framework provides its dependency injection system. Dependency injection is part of the inversion of control (IoC) principle. Dependency injection is a coding pattern in which a class does not create its dependencies, but instead, it is given from external source. In Angular, the dependency injection system provides declared dependencies to a class when that class is instantiated. [48]

The component that uses injected service does not need to know how to create that service. It is the job of the dependency injection system to create and manage dependencies. The component only needs to declare dependencies it needs. Service can be provided in the root module or any component. That means that service injected in that component and its child components is the same instance of that service. This enables the components to communicate together or use a shared resource.

### 3.2.2 StateManagerService

StateManagerService serves as a wrapper around NgRx Store. There is a reason for that. That is to provide future-proof modularity. So in the future, it will be easier to change Store implementation, because the rest of the application access Store via StateManagerService API, which would not change. Also, it is easier to control the flow of actions to the store and states from the store (e.g. using some filters).

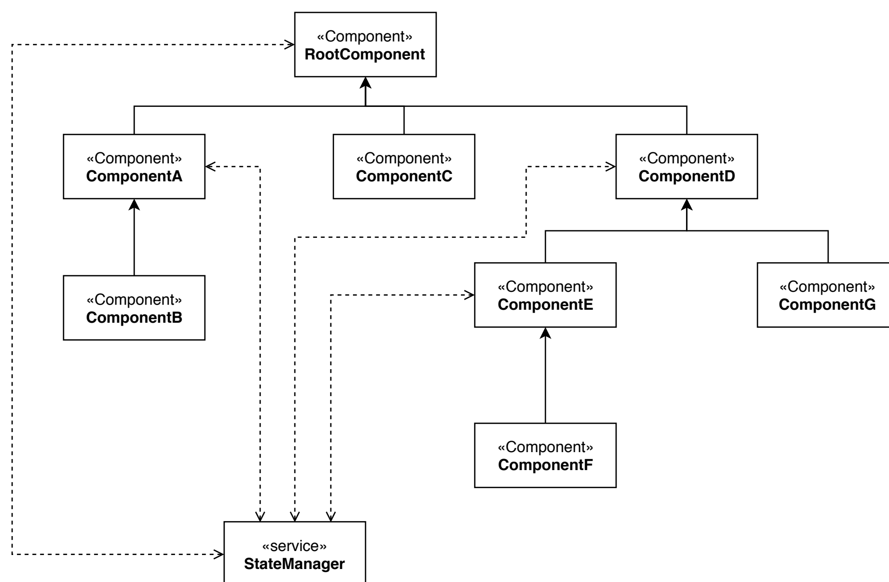



Figure 3.1: Purpose of Angular services

Because Angular and NgRx use RxJS, it is convenient and practical also to use it to implement a State Management subsystem. StateManagementService API can be divided into two parts: dispatching actions, which serves as input and state subscription, which serves as the output.

Method `dispatchAction(action: IAction)` receives action and pass it to the `actionStream$` observable. Actions from this stream are dispatched to the Store Module.

Store module provides root state as observable. Application state is organized as the state tree. Each feature or module of the application has its branch. Method `select()` is used to select only one branch of the state tree as `stateStream$` observable. Output methods create slices of the state from the state stream and return them as observables.

Each Angular component that needs to use state management has to implement `OnInit` and `OnDestroy` interfaces. That ensures that component implements `ngOnInit()` and `ngOnDestroy()` methods, which Angular calls when creating and destroying component and in which subscribing and unsubscribing happens. It necessary is to prevent memory leaks.

Effect and reducers for NgRx are provided in the Angular root module. The application also uses Redux DevTools, which is a debugging tool consisting of an Angular module and browser extension.[49]

```
@NgModule({
        declarations: [],
        imports: [
                StoreModule.forRoot(reducers),
                EffectsModule.forRoot([SignalrEffects]),
                StoreDevtoolsModule.instrument({
                        maxAge: 10,
                        serialize: true
                })
        ],
        providers: [],
        bootstrap: [],
        entryComponents: []
})
```

Listing 4: NgRx module initialization in Angular root module

#### 3.2.2.1 Effects

Effect are implemented in `PsiContractEditorEffects` class. It is an injectable class that gets injected to Store at startup. This class has two effects, synchronization which uses SignalrService to send action to the server, and logging, which logs all actions dispatched to the store in the console.

#### 3.2.2.2 Reducers

Reducer function is implemented as a static method of `PsiContractEditorReducer` class. The class serves as a wrapper around pure function. To mark the class as a static class, I used keyword abstract, that ensures that an instance of the class can not be created.

Reducer consist of a switch statement that decides based on the action type. That is a standard form of Redux reducers.

For every action that changes state in a way that influences the history of changes, reducer calls helper function `manageHistory`. The function manages undoStack and redoStack as well as updates versionId in the model.

Because the state must be immutable, change causes state object to be created. Creating a truly deep copy of an object is difficult in JavaScript. I used the option that is the easiest to implement: serialize an object to JSON and parse it back as a new object (containing the same data), with the use of standard JavaScript functions.

All action types are defined as public static read-only string value constants inside static class `ActionType` (implemented using the same trick as with reducer wrapper class). This class is used as an Enum class.

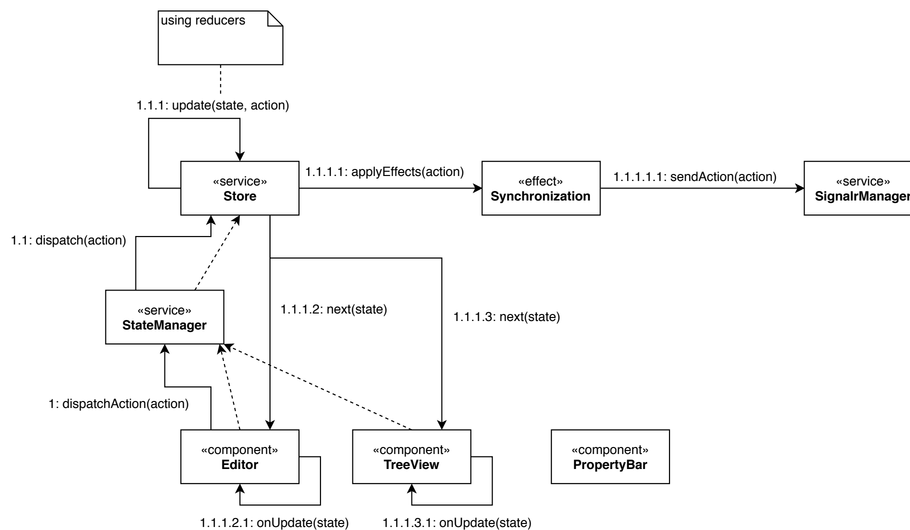



Figure 3.2: Communication diagram showing state change cycle
Dotted line represents subscription.

#### 3.2.2.3 Model

The model consists of two parts. First part is interfaces defining data structure and class implementing them. Classes do not implement any methods, only constructors. This part is in the model/model.ts file. The second part consists of a set of functions containing business logic. These functions are used to manipulate the model. This particular model is structured like a database, so most of these functions provide CRUD capability. There are also a few of type guards, which are functions for runtime type checking in TypeScript. This part is in model/utils.file.

### 3.2.3 SignalrService

SignalrService contains all logic that relates to the usage of the SignalR client library. When it is created, it builds and configures a connection to the server. To establish the connection using URI of SignalR hub on the server I implemented reconnecting mechanism, that makes only single attempt to make the connection during the startup of the application, but it tries to reconnect three times if the connection was lost after the initial connection. There is a delay between those reconnect attempts.

SignalrService has `sendAction` method, that is used by effects to invoke the method on the server. After the connection is established, a function that dispatches actions to the store is registered to the connection, to handle actions incoming from the server. This service also provides information about connection state (connected, disconnected, connecting) as public observable.

## 3.3 Server-side

Server-side part consist of ASP.NET Core application, build using C# and running on .NET Core framework.

- .NET Core version: 2.2.203
- ASP.NET Core version: 2.2.0
- ASP.NET Core SignalR version: 1.1.0

The server-side code has two important parts. First is the ActionHub, which is the SignalR hub. Hubs enable invoking methods from clients. The `DispatchAction` method is used to resend action to other clients. Type of action gets postfix *"_RECEIVED"* to indicate, that action will be received by the client. The other important part is Startup.cs file containing application settings including the setting of SignalR hub, the setting of port numbers, the setting of a path in the URI of the hub.

```
public class ActionHub : Hub
{
   public async Task DispatchAction(string type, string payload)
   {
       await Clients.Others.SendAsync("ReceiveAction",
          type + "_RECEIVED", payload);
   }
}
```

Listing 5: ActionHub - main part of server-side functionality

## 3.4 Usage

A working proof-of-concept is available at [50] and the open source code is placed in the public git repository at [51]

I used Redux DevTool to debug the application and to demonstrate its usefulness. The figure 3.3 depicts the Redux Devtools console during debugging. State of the applications is visualized in tree-like structure and history of processed actions is listed.

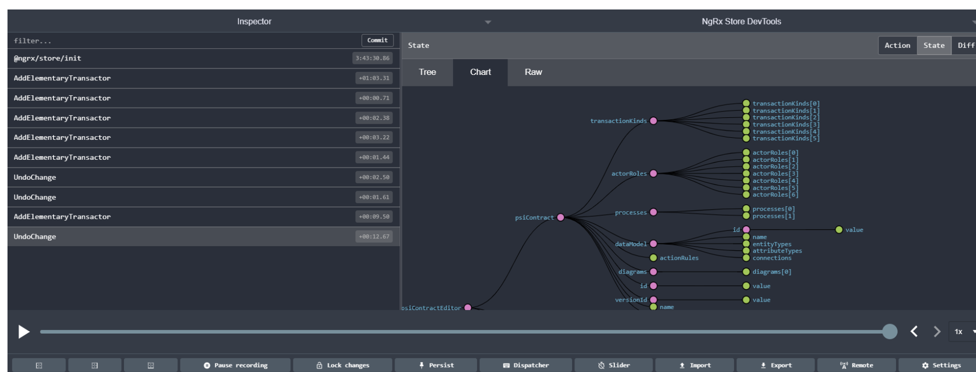



Figure 3.3: Redux DevTools console in action

When a user opens the application and edits a diagram in the modeling area, that component is generating and dispatching action based on the user input. The actions then cause the state to change, and that means, that other components receive the new state and render themselves accordingly.

In this case, that means that tree view on the left side that shows the structure of the model is always displaying the structure of a diagram in the modeling area. Moreover, because this client is connected to the server (which is shown by the indicator in the toolbar above), the state is synchronized. That means that all clients have precisely the same state and are rendered identically. This is shown in pictures 3.4 and 3.5.

## 3.5 Testing

One of the goals of Redux is to be easily testable. That applies to state management.

Because all data manipulation happens only in reducer and reducer is a pure function unit tests are easy to implement. This thesis contains a small set of unit tests of reducer function.

Testing SignalR communication is more difficult because it requires either to test server and client at the same time or to test one and replace the other by mock version. It is more integration testing than unit testing.

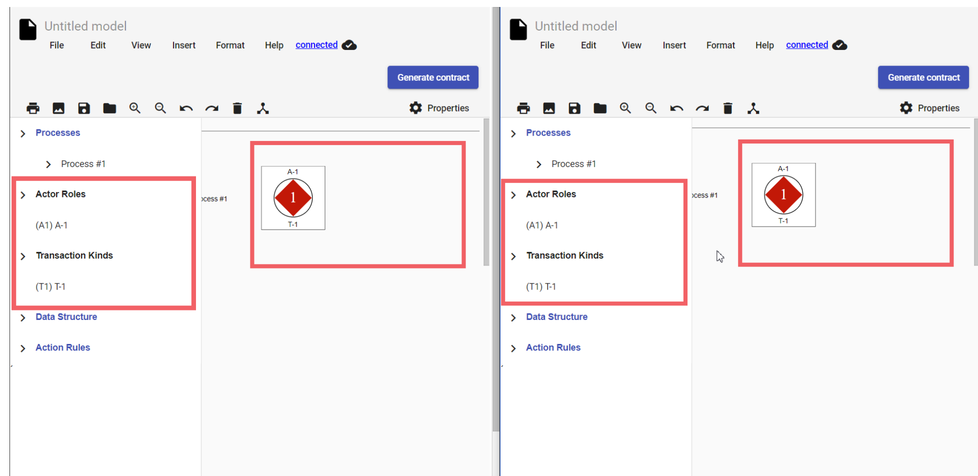



Figure 3.4: Synchronization between two clients opened in separate windows, part 1

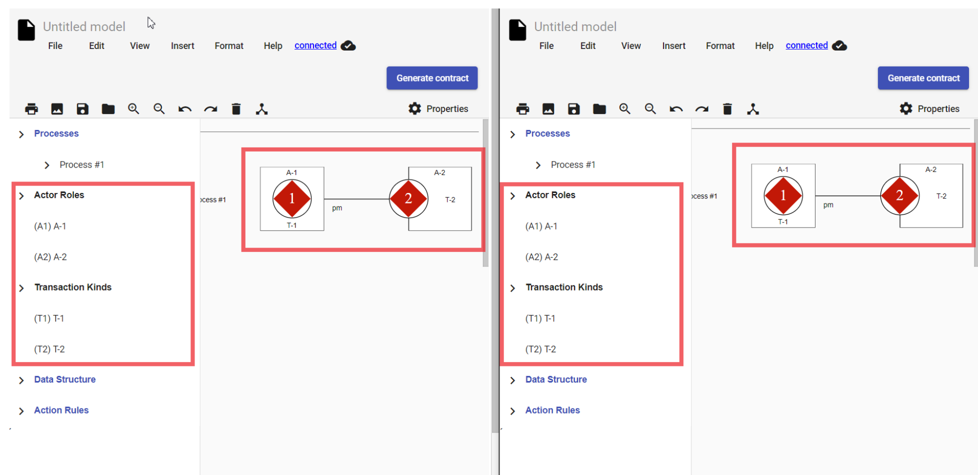



Figure 3.5: Synchronization between two clients opened in separate windows, part 2

Angular services and components right initialization can also be tested with unit tests.

## 3.6 Future Development Recommendation

The application works correctly, components dispatch action and get updated state, however further development is recommended.

The server should store application state and provide persistence, for example in a database. The server should also use state management system

like Redux. The application should have user accounts and synchronize data of individual users.

For implementing undo/redo feature can be used NgRx meta-reducers or third-party library. For creating a new state in reducers, some library for immutable objects can be used (e.g. Immer.js). For local machine persistence, some third-party library can be used with NgRx. Creating type guards could be automated, or runtime type checking library could be used.

Future development should also include more tests.

# Conclusion

The goal of this thesis was to review current techniques and approaches to state management and real-time synchronization in single page web applications and demonstrate their usage on google-drawings-like application.

The first part consists of a description of current design patterns and technologies as well as comprehensive reviews of most of the current approaches to this problem, using those patterns and technologies.

The second objective was to propose an architecture of state management and synchronization solution using Redux design pattern and SignalR library. A basic explanation of my design was shown in the implementation part of this thesis.

Implementation of proposed client-side state management solution is part of the DEMO methodology modeling tool available online at [50]. The full source code is released under the MIT license and is available in the public git repository at [51].

Because this is just proof-of-concept, in the future other features should be added to the state management system of the application. For example, to introduce user authentication and persistence to allow users to save their work on the server or use real-time synchronization feature while collaborating on a shared project.

# Bibliography


1. FACEBOOK. *Flux | Application Architecture for Building User Interfaces* [online]. Facebook Inc. [visited on 2019-04-09]. Available from: `https://facebook.github.io/flux/docs/in-depth-overview.html`.
2. ABRAMOV, Dan et al. *Motivation · Redux* [online]. 2018 [visited on 2019-05-06]. Available from: `https://redux.js.org/introduction/motivation`.
3. DAOUD, Fred. *Meiosis - Awesome State Management for Web Applications* [Awesome State Management for Web Applications] [online] [visited on 2019-04-09]. Available from: `https://meiosis.js.org/`.
4. WESTSTRATE, Michel et al. *Introduction | MobX* [online] [visited on 2019-05-08]. Available from: `https://mobx.js.org/index.html`.
5. *What is Vuex? | Vuex* [online] [visited on 2019-05-08]. Available from: `https://vuex.vuejs.org`.
6. BASAL, Netanel et al. *Introduction* [online]. 2018 [visited on 2019-05-08]. Available from: `https://netbasal.gitbook.io/akita`.
7. FETTE, I.; MELNIKOV, A. *The WebSocket Protocol* [Internet Requests for Comments]. RFC Editor, 2011. ISSN 2070-1721. Available also from: `http://www.rfc-editor.org/rfc/rfc6455.txt`. RFC. RFC Editor. `http://www.rfc-editor.org/rfc/rfc6455.txt`.
8. *Socket.IO - Overview* [online]. 2019 [visited on 2019-04-09]. Available from: `https://socket.io/docs/`.
9. MICROSOFT. *Introduction to ASP.NET Core SignalR* [online]. 2018 [visited on 2019-04-09]. Available from: `https://docs.microsoft.com/cs-cz/aspnet/core/signalr/introduction?view=aspnetcore-2.2`.
10. LANE, Kin. Overview of the backend as a service (BaaS) space. *API Evangelist*. 2015.

11. DEBILL, Erik. *Module counts* [online]. 2018. Available also from: http://www.modulecounts.com/.
12. VORBACH, Paul. *npm-stat.com* [online]. Available also from: https://npm-stat.com/.
13. SACHA GREIF, Raphael Benitte; RAMBEAU, Michael. *The State of JavaScript 2018* [online]. 2018. Available also from: https://2018.stateofjs.com/data-layer/overview/.
14. SAVKIN, Victor. Managing State in Angular Applications. *Nrwl*. 2017. Available also from: https://blog.nrwl.io/managing-state-in-angular-applications-22b75ef5625f.
15. MANCUSO, Sandro. *MVC, Delivery Mechanism and Domain Model* [online]. 2017 [visited on 2019-04-10]. Available from: https://www.javacodegeeks.com/2017/09/mvc-delivery-mechanism-domain-model.html.
16. BERNARD, Borek. Prezentační vzory z rodiny MVC. *Zdroják* [online]. 2009 [visited on 2019-05-08]. Available from: https://www.zdrojak.cz/clanky/prezentacni-vzory-zrodiny-mvc.
17. LUSSIER, D'Arcy. *MVVM Compared To MVC and MVP* [online]. 2009 [visited on 2019-05-04]. Available from: http://geekswithblogs.net/dlussier/archive/2009/11/21/136454.aspx.
18. MARCHAN, Flarnie. What is the Flux Application Architecture? *Medium* [online]. 2014 [visited on 2019-04-10]. Available from: https://brigade.engineering/what-is-the-flux-application-architecture-b57ebca85b9e.
19. SALCESCU, Cristian. An introduction to the Flux architectural pattern. *freeCodeCamp.org* [online]. 2019 [visited on 2019-05-13]. Available from: https://medium.freecodecamp.org/an-introduction-to-the-flux-architectural-pattern-674ea74775c9.
20. GELMAN, Ilya; DINKEVICH, Boris. *The Complete Redux Book: Everything you need to build real projects with Redux*. 2nd ed. Leanpub, 2018. ISBN 9789659264209.
21. NORING, Christoffer. *Architecting Angular Applications with Redux, RxJS, and NgRx: Learn to build Redux style high-performing applications with Angular 6*. Packt Publishing Ltd, 2018. ISBN 9781787122406.
22. RIGAU, Xavi. Introduction to Redux in Flutter. *Novoda blog* [online]. 2018 [visited on 2019-04-10]. Available from: https://blog.novoda.com/introduction-to-redux-in-flutter/.

23. BASAL, Netanel. Introducing Akita: A New State Management Pattern for Angular Applications. *Netanel Basal* [online]. 2018 [visited on 2019-05-08]. Available from: `https://netbasal.com/introducing-akita-a-new-state-management-pattern-for-angular-applications-f2f0fab5a8`.

24. RAM, Mohan. State Management in Angular using Akita. *Angular In Depth* [online]. 2018 [visited on 2019-05-08]. Available from: `https://blog.angularindepth.com/state-management-in-angular-using-akita-82f117d282dd`.

25. *MobX: Ten minute introduction to MobX and React* [online] [visited on 2019-05-08]. Available from: `https://mobx.js.org/getting-started.html`.

26. *Concepts & Principles | MobX* [online] [visited on 2019-05-08]. Available from: `https://mobx.js.org/intro/concepts.html`.

27. PURNELLE, Kevin. Handling React Forms with Mobx Observables. *RisingStack Engineering* [online]. 2016 [visited on 2019-05-08]. Available from: `https://blog.risingstack.com/handling-react-forms-with-mobx-observables`.

28. JAHR, Adam. Vuex Explained Visually. *Medium* [online]. 2018 [visited on 2019-05-08]. Available from: `https://medium.com/vue-mastery/vuex-explained-visually-f17c8c76d6c4`.

29. *Ajax* [online]. 2019 [visited on 2019-05-04]. Available from: `https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX`.

30. BHARATHVAJ, Ganesan. Polling vs SSE vs WebSocket— How to choose the right one. *codeburst* [online]. 2018 [visited on 2019-05-04]. Available from: `https://codeburst.io/polling-vs-sse-vs-websocket-how-to-choose-the-right-one-1859e4e13bd9`.

31. SALVET, Pavel. Komunikace v reálném čase díky Server-Sent Events a Web Sockets. *Interval.cz* [online]. 2015 [visited on 2019-05-04]. Available from: `https://www.interval.cz/clanky/komunikace-v-realnem-case-diky-server-sent-events-a-web-sockets`.

32. *The WebSocket API (WebSockets)* [online]. 2019 [visited on 2019-05-04]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`.

33. MÁJSKÝ, Michal. *Analýza současných řešení Backend-as-a-Service pro vývoj mobilních a webových aplikací.* Praha, Česká republika, 2016. Master's thesis. České vysoké učení technické v Praze.

34. MÁJSKÝ, Michal. Začínáme s Backend as a Service. *Zdroják* [online]. 2016 [visited on 2019-04-10]. Available from: `https://www.zdrojak.cz/clanky/zaciname-s-backend-service/`.

35. *Backend-as-a-Service* [online]. Cloudflare Inc. [visited on 2019-04-10]. Available from: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/.

36. *Backend as a Service* [online]. Backendless Corp. [visited on 2019-04-10]. Available from: https://backendless.com/platform/backend-as-a-service/.

37. BELGIUM, Ordina. *NGRX vs. NGXS vs. Akita vs. RxJS: Fight! - Orjan De Smet* [online]. 2018 [visited on 2019-05-08]. Available from: https://ordina-jworks.github.io/angular/2018/10/08/angular-state-management-comparison.html.

38. WHITFELD, Mark. NGXS State Operators. *Medium* [online]. 2019 [visited on 2019-05-13]. Available from: https://medium.com/ngxs/ngxs-state-operators-8b339641b220.

39. *NGXS - Introduction* [online]. 2019 [visited on 2019-05-13]. Available from: https://ngxs.gitbook.io/ngxs.

40. MOHL, D. *Building Web, Cloud, and Mobile Solutions with F#: Create Scalable Apps with ASP.NET MVC 4, Azure, Web Sockets, and More.* O'Reilly Media, 2012. ISBN 9781449333720. Available also from: https://books.google.cz/books?id=MH3-T2jGFsEC.

41. MACKEY, A.; TULLOCH, W.S.; KRISHNAN, M. *Introducing .NET 4.5.* Apress, 2012. Apressus Series. ISBN 9781430243328. Available also from: https://books.google.cz/books?id=I-dHxFjfbi8C.

42. VYAS, Rahul. Firebase Cloud Firestore v/s Firebase Realtime Database. *Medium* [online]. 2017 [visited on 2019-05-14]. Available from: https://medium.com/@beingrahul/firebase-cloud-firestore-v-s-firebase-realtime-database-931d4265d4b0.

43. MERENYCH, Sofiya. Overview of mobile backend as a service providers. Which one will be your choice? [online]. 2019 [visited on 2019-05-14]. Available from: https://clockwise.software/blog/overview-of-mobile-backend-as-a-service-providers.

44. ALL IN MOBILE. Backendless. *Medium* [online]. 2016 [visited on 2019-05-14]. Available from: https://medium.com/all-in-mobile/backendless-txt-a0ee1c2bf16c.

45. DANILOVA, Olga. *Five things you cannot do with Firebase (but can with Backendless) | Backend as a Service Platform* [online]. 2018 [visited on 2019-05-14]. Available from: https://backendless.com/five-things-you-cannot-do-with-firebase-but-can-with-backendless.

46. SKOTNICA, Marek. *Lecture notes in Software Team Project: Functional Specification - PSI Contract Designer.* 2019.

47. MICROSOFT. *TypeScript* [online]. 2019 [visited on 2019-05-13]. Available from: https://github.com/Microsoft/TypeScript.
48. GOOGLE. *Angular - Architecture overview* [online]. 2019 [visited on 2019-05-13]. Available from: https://angular.io/guide/architecture.
49. *Extension · Redux DevTools Extension* [online] [visited on 2019-05-13]. Available from: http://extension.remotedev.io/.
50. CCMI RESEARCH GROUP. *Enterprise Designer* [online] [visited on 2019-05-13]. Available from: https://smartcontracts.azurewebsites.net/.
51. CCMI RESEARCH GROUP. *Enterprise Designer Repository* [online] [visited on 2019-05-13]. Available from: https://dev.azure.com/ccMiResearch/_git/VSContract.

Appendix A

# Acronyms

**AJAX** Asynchronous JavaScript And XML

**API** Application Programming Interface

**BaaS** Backend as a Service

**CQRS** Command Query Responsibility Segregation

**CRUD** Create, read, Update and Delete

**DEMO** Design & Engineering Methodology for Organizations

**GUI** Graphical User Interface

**HTML** Hypertext Markup Language

**HTTP** Hypertext Transfer Protocol

**HTTPS** Hypertext Transfer Protocol Secure

**IoC** Inversion of Control

**JSON** JavaScript Object Notation

**MBaaS** Mobile Backend as a Service

**MVC** Model-View-Controller

**MVP** Model-View-Presenter

**MVVM** Model-View-ViewModel

**OOP** Object-Oriented Programming

**RCP** Remote Procedure Call

**REST** Representational State Transfer

**RxJS** Reactive Extensions for JavaScript

**SDK** Software Development Kit

**SQL** Structured Query Language

**SSE** Server-Sent Events

**TCP** Transmission Control Protocol

**UI** User Interface

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**XML** Extensible Markup Language

# Appendix B

# Contents of enclosed CD

```
├── readme.txt ........................ the file with CD contents description
├── src ........................................ the directory of source codes
│   ├── application ................................ implementation sources
│   │   └── thesisfiles.txt . list of source code files belonging to this thesis
│   └── thesis .............. the directory of LaTeX source codes of the thesis
└── text ........................................... the thesis text directory
    └── thesis.pdf ......................... the thesis text in PDF format
```