React - uniform management of state and time

Jukka Marttinen
The purpose of this thesis is not to introduce the full ecosystem needed to build applications in ReactJS but React itself and how to manage the State in React. React is not a full-blown framework and instead it is just a library with a single concern, that is the V (view) in MVC (Model View Controller).

The objectives of this thesis theory part, is to shed some light into what is needed to develop React in the context of State management. What is React and what is it good for? What do you need to know about React’s own local State management or third-party State management libraries?

To adhere the research questions, a study application was made. This application was contracted by HiQ for a Customer company, not to be disclosed in this thesis as a non-disclosure agreement was made with the Customer. This application was made with a high importance of State management within the requirements.

This thesis goes through the phases of JavaScripts history and early development, which leads up from SPA (Single Page Applications) frameworks, to the current phase of modularity pattern as the most popular pattern of Client-side development.

The history of React and the key points of React ecosystem, such as JSX, VirtualDOM and its local State management through modular patterns and architecture have become something of a standard in the Client-side developers community and through, such the management of State has moved towards the Client-side from Server-side.

Flux is more of a pattern than a framework that was developed by Facebook, as an opinionated way to manage the State in unidirectional way.
The emergence of Flux left the React community in disarray with multiple Flux implementations with their own way of implementing Flux pattern with incompatibilities between them.

Redux came to the scene as Flux like library, with opinions how to manage the state with Functional programming paradigm, State-tree, Time-travel and HMR (Hot-Module-Reloading). Redux came as the de-facto tool how to manage the State in an opinionated way and the de-facto status of it stands to this day.

MobX turned away from Flux in a hole hearted way with Reactive Functional Programming. The MobX library abstracts State management with reacting to changes in the data with traditional OOP concepts, which cut down the demanded boilerplate compared to other State management libraries. MobX achieves of being efficient in performance and time to develop by cutting down the time to develop applications.

MobX-state-tree is a union of two paradigms of mutable and immutable. It is an extension library to MobX with more opinion of How than the parent library. An application was made as a proof-of-concept application to a Customer, to determine, if the MST library could be an applicable replacement option to Redux or MobX. The application was done following a loose agile development cycles, determining Customers needs. The application done was a true FullStack JavaScript application where the Client / Backend shared the same programming language and even the CSS styles were done with CSS-in-JS (CSS with JavaScript).

MST accomplishes to structure the application with clear opinionated patterns and combines the best of the immutability and mutability paradigms. The libraries fate is sealed with the advancement of its parent library that is MobX. However, MST complements MobX library in a way that (in this thesis writers opinion) will add significant value to HiQ and its Customers.

**Keywords**
React ecosystem, State management, Flux, Redux, MobX, MobX-state-tree
1 Introduction ...........................................................................................................................................1
2 Research question, objectives and scope ..........................................................................................2
  2.1 Research question .............................................................................................................................2
  2.2 Scope and objectives ..........................................................................................................................3
3 History in making ..................................................................................................................................4
  3.1 The history of JavaScript and why it took so long of it to become dominant .........................4
  3.2 History of React ..................................................................................................................................6
4 JSX sugar on top of JavaScript .........................................................................................................8
5 DOM vs VirtualDOM ............................................................................................................................11
  5.1 DOM ..................................................................................................................................................11
  5.2 VirtualDOM ....................................................................................................................................12
      5.2.1 How the actual DOM works ......................................................................................................12
      5.2.2 How the VirtualDOM works ....................................................................................................13
6 Building the VirtualDOM with ReactElements and user defined Components .........................14
7 The M and C to complement the V .....................................................................................................16
  7.1 React component state .....................................................................................................................17
8 Flux an architecture to flow the data one way ...............................................................................20
  8.1 Flux implementations .......................................................................................................................21
9 Redux to predict the state in an opinionated way ............................................................................23
  9.1 Why Redux .......................................................................................................................................23
  9.2 Redux loves Functional Programming ............................................................................................24
  9.3 Redux state tree ..................................................................................................................................25
  9.4 Redux API core concepts ..................................................................................................................26
  9.5 Async Redux .....................................................................................................................................27
10 MobX for less complexity ................................................................................................................29
  10.1 Avoid staleness through Reactivity ...............................................................................................29
  10.2 The gist of three behind MobX .....................................................................................................30
      10.2.1 The Observer ............................................................................................................................31
      10.2.2 Explicitly actions ........................................................................................................................32
      10.2.3 Derive all ....................................................................................................................................32
11 Project Research plan .........................................................................................................................34
  11.1 Background ......................................................................................................................................34
  11.2 Research Scope and Objectives ....................................................................................................34
  11.3 Work described ...............................................................................................................................35
12 Design ..................................................................................................................................................36
  12.1 User stories .......................................................................................................................................36
      12.1.1 User story for the Administrator ............................................................................................36
      12.1.2 User story for the Electrician ..................................................................................................36
  12.2 Use Cases ......................................................................................................................................37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2.1 Use Case Login</td>
<td>37</td>
</tr>
<tr>
<td>12.2.2 Use Case for Fast Command</td>
<td>38</td>
</tr>
<tr>
<td>12.2.3 Use Case for Command Request with a defined type</td>
<td>39</td>
</tr>
<tr>
<td>12.2.4 Use Case search and filter history requests</td>
<td>40</td>
</tr>
<tr>
<td>12.3 Layout wireframe</td>
<td>42</td>
</tr>
<tr>
<td>13 Application build with MobX-state-tree</td>
<td>46</td>
</tr>
<tr>
<td>13.1 Development Tools</td>
<td>46</td>
</tr>
<tr>
<td>13.2 Application stack</td>
<td>46</td>
</tr>
<tr>
<td>13.2.1 Backend stack</td>
<td>46</td>
</tr>
<tr>
<td>13.2.2 Frontend stack</td>
<td>46</td>
</tr>
<tr>
<td>14 Brief theory behind the concepts of MST</td>
<td>49</td>
</tr>
<tr>
<td>14.1 State-tree</td>
<td>49</td>
</tr>
<tr>
<td>14.1.1 Structural sharing of State-tree</td>
<td>50</td>
</tr>
<tr>
<td>14.2 Time travel through Snapshots and patches</td>
<td>53</td>
</tr>
<tr>
<td>14.3 Typesystem forming stores</td>
<td>53</td>
</tr>
<tr>
<td>15 Application made with MST</td>
<td>55</td>
</tr>
<tr>
<td>15.1 Models are stores or stores are models</td>
<td>55</td>
</tr>
<tr>
<td>15.1.1 AuthenticationStore</td>
<td>55</td>
</tr>
<tr>
<td>15.1.2 CommandStore</td>
<td>57</td>
</tr>
<tr>
<td>15.1.3 queueStore</td>
<td>58</td>
</tr>
<tr>
<td>15.1.4 HistoryStore</td>
<td>59</td>
</tr>
<tr>
<td>15.2 Views</td>
<td>60</td>
</tr>
<tr>
<td>15.2.1 Index.js as the entry of the application</td>
<td>60</td>
</tr>
<tr>
<td>15.2.2 Login Component</td>
<td>62</td>
</tr>
<tr>
<td>15.2.3 MainContainer Component to hold the views of the application</td>
<td>62</td>
</tr>
<tr>
<td>15.2.4 Editing a Command</td>
<td>64</td>
</tr>
<tr>
<td>16 Conclusion</td>
<td>66</td>
</tr>
<tr>
<td>16.1 Development with MobX-State-Tree</td>
<td>66</td>
</tr>
<tr>
<td>16.2 Testability of MobX-State-Tree</td>
<td>67</td>
</tr>
<tr>
<td>16.3 MobX and MobX-State-Tree community support</td>
<td>67</td>
</tr>
<tr>
<td>16.4 Added value HiQ and to the Customers</td>
<td>68</td>
</tr>
<tr>
<td>16.5 Outcomes of learning</td>
<td>68</td>
</tr>
<tr>
<td>17 Summary</td>
<td>70</td>
</tr>
<tr>
<td>References</td>
<td>72</td>
</tr>
<tr>
<td>Appendices</td>
<td>81</td>
</tr>
<tr>
<td>Appendix 1. The Project structure on root level</td>
<td>81</td>
</tr>
<tr>
<td>Appendix 2. The Client root structure</td>
<td>81</td>
</tr>
<tr>
<td>Appendix 3. The Client Component structure</td>
<td>82</td>
</tr>
<tr>
<td>Appendix 4. The Client Model / Stores structure</td>
<td>83</td>
</tr>
</tbody>
</table>
## Terms and abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Client application state</td>
</tr>
<tr>
<td>ReactJS</td>
<td>JavaScript View library</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Markup Language</td>
</tr>
<tr>
<td>XML</td>
<td>Data format shortened from Extensive Markup Language</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Interpreted Multiparadigm Language that works in multiple platforms</td>
</tr>
<tr>
<td>EcmaScript</td>
<td>JavaScript standard</td>
</tr>
<tr>
<td>Babel</td>
<td>Transpiler / compiler for JavaScript super-sets that add functionality to the basic JavaScript language</td>
</tr>
<tr>
<td>Webpack</td>
<td>JavaScript and CSS bundler</td>
</tr>
<tr>
<td>SPA</td>
<td>Single Page Application</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Line-by-line Computing usually with multiple threads</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Non-blocking parallel execution on a single thread</td>
</tr>
<tr>
<td>AJAX</td>
<td>Asynchronous JavaScript and XML</td>
</tr>
<tr>
<td>JSON</td>
<td>Data format shortened from JavaScript Object Notation</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller Architectural pattern</td>
</tr>
<tr>
<td>NodeJS</td>
<td>Server-side non-blocking IO that uses the Googles JavaScript V8 engine</td>
</tr>
<tr>
<td>NPM</td>
<td>Node Package Manager that is used to import OpenSource third party JS libraries to a project</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model for accessing and updating the HTML</td>
</tr>
<tr>
<td>VirtualDOM</td>
<td>Virtualized and abstracted copy of the real DOM</td>
</tr>
<tr>
<td>JSX</td>
<td>Superset to JavaScript</td>
</tr>
<tr>
<td>XSS</td>
<td>Cross-Site-Scripting attack</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Orientated Programming</td>
</tr>
<tr>
<td>Concept</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Imperative Programming</td>
<td>Programming paradigm where the state of the program is changed in sequential procedures</td>
</tr>
<tr>
<td>Functional Programming</td>
<td>Programming paradigm where pure functions are used by avoiding shared state, side-effects and mutation of data</td>
</tr>
<tr>
<td>Unidirectional data flow</td>
<td>Data passed explicitly from parent element to child element</td>
</tr>
<tr>
<td>Flux</td>
<td>Pattern to handle unidirectional data flow</td>
</tr>
<tr>
<td>Redux</td>
<td>State management library following Flux and Functional Programming</td>
</tr>
<tr>
<td>MobX</td>
<td>State management library following OOP and Functional Reactive Programming</td>
</tr>
<tr>
<td>Mutable Object</td>
<td>Object that can be changed after creation</td>
</tr>
<tr>
<td>Immutable Object</td>
<td>Object that cannot be changed after creation</td>
</tr>
<tr>
<td>Snapshot</td>
<td>State in specific point in time</td>
</tr>
<tr>
<td>State tree</td>
<td>Immutable object representing application state</td>
</tr>
<tr>
<td>API</td>
<td>Application Interface</td>
</tr>
<tr>
<td>DB</td>
<td>Data Base</td>
</tr>
<tr>
<td>REST Api</td>
<td>Architectural pattern that is based upon HTTP protocol. Abbreviation to Representational State Transfer.</td>
</tr>
</tbody>
</table>
1 Introduction

The purpose of this thesis is not to introduce the full ecosystem needed to build applications in ReactJS but React itself and how to manage the State in React. React is just a View library and not a full-blown mature framework, such as AngularJS. Working with React the developers need to fill in the holes left in the flow and control of data by building a framework themselves in modular way, brick by brick. React has patterns that are very opinionated in the flow of the data and best practices needed to optimize the applications in the right way. This thesis will introduce some of the basic understanding of the building blocks, which make React itself and then open up with some of the most tools used in developing with React.

React in itself has a pretty manageable learning curve, if you have basic knowledge about such subjects as HTML, XML and JavaScript. But knowing about just React is not in itself enough to tackle the task of creating full scale web applications. This thesis is about solving the issue of developing full scale web applications with React and especially how to manage the state within the ecosystem.

The study behind this thesis was contracted by HiQ Finland. The writer of this thesis works at HiQ Finland as an FullStack developer / IT-Consultant. HiQ Finland is a ICT-Consultant firm with its head quarters located in Otaniemi Espoo Finland. HiQ Finland is a daughter company of HiQ, which is one of the leading ICT-Consultant firms in North-Europe with 1600 employees in six countries. (HiQ, about us 2018)
2 Research question, objectives and scope

2.1 Research question

The world of JavaScript frameworks has exploded exponentially, introducing new frameworks constantly with the rise of NodeJS and its package manager NPM. In this vast expanding sea of new NPM packages and recycled or new programming paradigms, developers and thus companies might easily drown, when going fishing for the right resources. For resources needed to develop products for customers or end users in general. Every couple of months there seems to surface a new JavaScript framework or library on to the scene with a new model for programming to rule them all. Package managers such as NPM or equivalent allow the users to develop and consume modules or packages in development with ease by downloading them and adding references to them in the programs source code. Packages which are JavaScript libraries or frameworks are being consumed and learned by the developers on a more increasing pace and this does not seem to slow down anytime soon. (NPMJS DOCUMENTATION 2017, What is NPM).

ICT-industry has dubbed a new catchphrase for this phenomenon and it is quite figurative. The new catchphrase is JavaScript fatigue. The winners against this fatigue are the ones who adopt the best practices early and tackle them with simplicity, productively and thus hopefully producing quality along the way in the process.

The objectives of this thesis are to shed some light into what is needed when developing with React in the context of State management. What is React and for what kind of development is it good for? Resources needed for development considering State management. What do you need to know about React’s own local State management or third-party State management libraries? This thesis focuses on React’s local State and on three main libraries, which two of them are introduced in the theory part and the third is left to be the source for this thesis study.

How did the need for React, and in particular for its State management evolve and what does State management look like now and where is it heading? These are the questions that this thesis tries to answer.
2.2 Scope and objectives

The scope of this thesis is to go through the theory regarding React and the linking scope of State management. Firstly, React and its main ideas behind it presented. React is a modular library, which tries not adhere fully to the question of State management and the fulfilment of that prerogative is left to the developers. This thesis goes through and compares the most popular paradigms of State management and adds a contender to the mix in the study part.

To address these questions an application was made as a study application. This application was contracted by HiQ for a Customer company with a high importance of State management required. This application and its State management library usage is gone through and explained in the study part of this thesis. The study part of this thesis is under need to know basis and under the rules of confidentiality. A commissioning has been signed with commissioning party HiQ with the agreement of the Customer in question, which receives the study application as a production ready application ordered by the Customer.
3 History in making

It is often said that React is just the V in MVC and React on its own is nothing but a JavaScript library and not a full-blown JavaScript Framework. On being just a small library React achieves on being declarative in its nature, efficient on performance and flexible in development time. ReactJS is meant for creating user interfaces on web sites, mobile applications with ReactNative and in the future it will allow desktop development with React Desktop. ReactVR has risen to the call concerning virtual reality. (ReactJS Documentation 2017, Tutorial).

3.1 The history of JavaScript and why it took so long of it to become dominant

It is always good to know a little bit of background history when it comes to the topic in question. Such as how did JavaScript evolve to be what it is today and how did it evolve to the present pinnacle that is ReactJS? The meaning of pinnacle taking as loosely as possible in this context. How did JavaScript evolve to SPA's (Single Page Applications) along the way from full out frameworks to modular libraries that complement each other?

JavaScript got its very humble start at the mid 90's at Netscape by the developer Brendan Eich. The beginning of JavaScript had a lot of confusion from the very beginning, starting with the naming convention of it. First dubbed Mocha, then LiveScript. Javascript ended up combining the coffee synonym beginning Mocha to the obvious ending Script. The obvious benefit was to ride the popularity of another coding language, namely Java as a marketing scheme. (JavaScript; Simon Timms, 2014. Mastering JavaScript Design Patterns, 8) JavaScript borrowed more than just a name from Java, but the main difference between them stays to this day. This difference is that JavaScript is an interpreted language instead of compiled language that Java is. Interpreted language means that there must be a program that reads the language / interprets it line by line and runs it on the fly in its program environment. In the beginning this dynamically / loosely typed interpreted language was meant only to run on the client-side, in the browser and it was received as a gimmick by many of the target audience members, meaning software developers. (Stanford University Site2017, Introduction to JavaScript; Simon Timms, 2014. Mastering JavaScript Design Patterns, 8)

Microsoft made their own implementation of JavaScript named JScript and implemented that to their main browser implementation at the time. This browser implementation was the Internet Explorer. This lead to competitive implementations of JavaScript and contributed to most of the problems we face today with cross-browser incompatibilities. Pretty soon, after the great divide, NetScape handed out the governance of JavaScript to ECMA
For years, the deviations between browsers and especially the emergence of the winner of browser wars, which was in the end the infamous Internet Explorer, meant that most of the time developing with JavaScript would be difficult and time consuming. One of the reasons being that developers would need to determine the running browser from the start of the execution. This resulted in a code that was dubbed as “spaghetti code”. This code was difficult to read / write, it was error prone and quite frankly a lot of times was not deemed worthy of the implementation effort. It was not until the emergence of JQuery or equivalent libraries, which gave cross-browser compatibility to DOM manipulations, that JavaScript became more widely adopted in the development community. The second major thing that aided the rising popularity of JavaScript was the rising of AJAX (Asynchronous JavaScript and XML). AJAX was developed back in the year 1999 but did not get widely known until it was used in Google’s Gmail back in 2004. The tipping point was finely achieved with this application, as it showed the true dimensions that could be achieved with JavaScript using AJAX techniques. With AJAX you could update the application without reloading the browser window, fetch and push data to the server quietly in the background without interruptions. (W3Schools Site 2017, Ajax introduction) The applications made with the holy trinity that is JavaScript, AJAX and JSON (JavaScript Object Notation) are called SPA’s (Single Page Applications). You could from the very beginning make SPA applications with vanilla JavaScript but not without twisting your fingers out of place from the effort. Frameworks such as Angular, Ember and Backbone game to the frontstage to solve this problem with their own abstractions through architectural pattern called MVC (Model View Controller). From the MVC frameworks the most notable one to come out on top as the industry de-facto standard for a time was AngularJS. Angular uses two-way data binding between the model and the view to eliminate the expensive DOM manipulation. (AngularJS Documentation 2017, Main page; JavaScript; Simon Timms, 2014. Mastering JavaScript Design Patterns, 9-11)

JavaScript has now become widely used partly, because of instead of running just on the targeted device as native bytecode, like compiled languages do, JavaScript can run on every device as a cross-platform interpreted language. The native implementations of JavaScript on server, mobile, VR (Virtual Reality) and even on desktop are becoming more common. The main reason behind this, is the rise of modular JavaScript paradigm with
NodeJS, NPM and ReactJS. JavaScript has become the most popular open source language out there, with the help of NPM and Github. (Techbeacon Site 2016, Programming language rankings; Standford University Site 2017, Introduction to JavaScript; Simon Timms, 2014. Mastering JavaScript Design Patterns, 13-15)

3.2 History of React

The origin of React can be traced back to Facebook’s PHP extension XHP, which was meant to minimize the possibility of XSS attacks (Cross-Site-Scripting attack). XSS attacks are initiated by users with malicious intents by embedding JavaScript in the input fields as a content of the form. By this attack method, the malicious user tries to steal data or otherwise compromise parts of the system or the entire system itself. The main concern of XHP was to prevent these XSS attacks and with the usage of it lead to another problem. In Dynamic Web Applications the Client talks to the correspondent Server with a roundtrip of request and response. Every proceeded request and response cycle takes time and this time shows up to the user as a response time lag. The use of XHP in Facebook increased the amount of roundtrips to the Server and thus decreased the responsiveness of the application. This was unacceptable and lead to the demise of XHP and in the aftermath to the birth of ReactJS. React in many ways borrows from XHP and is greatly indebted to it. (THE NEWSTACK Site 2014, JavaScript history and why it led to ReactJS)

When Facebook engineers started building React they decided to scrap the idea of building yet another bulky “all inclusive” MVC framework, which tries to answer all of the questions of how. Facebook instead decided to go with the principles of modular programming, concentrating only on the View layer off MVC and left up to developers themselves to figure out how to build the rest. Thus, this contributed a lot to the emergence of modularity becoming essential for frontend developers of today. Even though modularity existed even before, third party packages still answered only the most simple questions in JS applications. Questions behind, such things, as validation, animations etc. The more complicated questions, such as the management of State was kept firmly in the hands of frameworks themselves. (ReactJS Documentation BLOG 2013, Why React)

One benefit of React not being a full-blown JavaScript framework is that it avoids the usual fault of typical JavaScript frameworks, which abstract the JavaScript language itself. Without the annoyance of abstraction, you can get the full power and flexibility of the JavaScript language. This means that the developers can build up the solutions with abstractions themselves from the ground up or they can choose what architecture, pattern or styles they want to implement themselves. React is opinionated in a lot of things and one
thing is that it applies modularity as one of its core concepts. Modularity in React is introduced by Components, which are the main building blocks of a React application. Components are React classes that are user defined, reusable and, which from the application is built upon. Every Component combines its logic and mark-up and thus it makes the code more readable and easier to extend later on. The aim of Facebook was to avoid making just another templating language by letting HTML and JavaScript co-exist within the same file. (ReactJS Documentation BLOG 2013, Why React)
4  JSX sugar on top of JavaScript

JSX goes often hand in hand in writing React applications and is the only major abstraction that Facebook introduces with React. Facebook developed this JavaScript extension alongside with React to ease the development process with React. React abstracts JavaScript with JSX into an XML / HTML like syntax. Even if the element resembles basic HTML or XML, it is in fact a JavaScript function and all of the JSX elements map to basic JavaScript functions in the process of transpiling. (Facebook Github Site 2017, JSX; Facebook Github Site 2017, JSX)

JSX is used for producing React elements and user defined components. The rough difference between elements and components are that Elements are abstracted HTML elements and Components are user defined elements. Elements are written in lowercase and components with uppercase. This is to hint React about the difference between them and it introduces optimized performance. (ReactJS Documentation 2018, JSX in depth)

JSX is not be implemented by the browsers themselves or any other JS engines as it is not to be meant as a EcmaScript proposal. JSX is just an added feature, plugin or super-set to EcmaScript. Transpilers are meant to be used with JSX to transform it to plain JavaScript / EcmaScript.

React is secure by default against XSS (Cross-Site-Scripting) attacks. (Singsys Blog Site 2017, Basic XSS vulnerability in ReactJS) The origin of React and JSX can be traced back to Facebook’s PHP extension XHP, which was meant to minimize the possibility of XSS attacks. (THENEWSTACK Site 2014, JavaScript history and why it led to ReactJS) ReactDOM escapes the values that are embedded in JSX by the user. All inputs are converted to Strings before the render method occurs and this means that you cannot execute any ordinary XSS attacks. (ReactJS Documentation 2017, Introducing JSX) This additional security layer baked in out of the box with React makes it a considerable ally when writing Client-side code in comparison to competitor frameworks, which may lag this kind of default security and the developers themselves are responsible for keeping the apps secure.

JSX does not acquire a runtime library and it is just a pre-processor step, which you can use to add XML syntax for JavaScript. This has the advantage of being more meaningful semantically. It helps with the understanding of the code, as the elements are written so that they form tree structures, just like normal HTML templating. This is more familiar for
all of those, who are not familiar with JavaScript but are familiar with HTML and XML. JSX makes it easy for other developers to understand your code more quickly without a long introductory processes deep diving to the code itself. (Jomendez Site 2018, JSX preprocessor syntax)

Developer can choose to drop JSX out of development cycle but that leaves the developer out of the benefits that come with JSX, such as speed, type-safety, security and the benefit of being more declarative than just vanilla JavaScript itself. Speed comes with the optimization of produced code with the compilation / transpilation of JSX to JavaScript. Most of the errors are caught at compilation time, adding type-safety to the mix. Being more declarative means the fact that JSX is more XML / HTML like and that gives familiarity, structure and speed in development phase by being able to form tree like structuring to the developed code. (TutorialsPoint Site 2017, ReactJS / JSX)

Babel is used to transpile / transform JSX to JavaScript. Babel is not the only transpiler out there for JSX, but it is the most well-known of them. Transpilers are here to stay in the modern JavaScript development process, due to the yearly releases of ECMA standards that has been agreed upon. PureScript, CoffeeScript and TypeScript, which are like different JavaScript dialects, ship out with their transpilers packaged alongside of them. Babel is best used with a JavaScript bundler such as WebPack or Browserify or it can be injected as a separate Script tag. However injecting Babel as a script tag is not recommended for larger scale apps. (BabelJS Site 2018, What is Babel; ReactKunfu Site2015, Guide to modern JS Tooling)

Babel enables the developer to transpile not just only JSX to JavaScript, but ES6 or higher versions of EcmaScript as well. ES6 (or ES2018 (ES9) the newest version) is the new version / release of JavaScript, and it helps with the development process enormously, as it introduces a working set of improvements to the language. (W3Schools Site 2018, JS Versions)

The name of JavaScript is widely known and adopted common alias for ECMAScript. JavaScript was submitted for standardization to ECMA international. ES9 or ECMAScript 9 is the new iteration of ECMAScript standard. ES9 is also known as ES2018 and even renamed as such, as the standardization process is moving towards yearly releases with promises of new specifications. (Benmccormick Site 2015, JavaScript versioning)

The usage of Babel with development brings out the benefits of using ES9. Babel transpiles the code made in ES9 to ES5 for the reason that modern web browsers do not fully support the new language specifications yet. ES6 was the long awaited major release
since 2009 when ES5 came out. ES6 uses the arrow syntax for shortening the function declaration. This functionality has been seen before in other languages, such as C#, which incorporated lambda function calls in LINQ (Language Integrated Query) (OdeToCode Site 2014, Arrow functions in ES6) early on and the new version of Java, which is Java8 has also adopted this (Benmccormick Site 2015, JavaScript versioning). In ES6 the functions declared in as arrow functions are always declared as this inside a class as they bind automatically to the class. Arrow functions allow easier handling of scope, for example: the child functions, which live inside of the parent function will share all of the parent functions arguments / parameters. (Medium Site 2018, Arrow function as class property) All the versions of EcmaScript go through five phases before they can be added to the mature language as specifications. These phases are: Strawman, Proposal, Draft, Candidate and the final version, which is called Finished. (tc39 GitHub Page 2018, Ecma version Phases; Benmccormick Site 2015, JavaScript versioning)
5 DOM vs VirtualDOM

The adaption of VirtualDOM is an essential part in Facebooks thinking of making something new with ReactJS. VirtualDOM is a very lightweight “virtual / fake” DOM that is an in-memory implementation of the real DOM (Document Object Model). VirtualDOM is a cross-browser implementation, that is meant for fast and efficient DOM manipulation. VirtualDOM however can work cross-platform in environments such as mobile or desktop and is not limited into working just in the browser. (O’Reilly Site 2014, The secrets of React’s VirtualDOM)

5.1 DOM

The Document Object Model is a language-neutral API, standardized by W3C. DOM is used for programs and scripts to access and update the HTML document in a dynamic way. There are three kinds of standards of the DOM, which are: Core DOM, XML DOM and HTML DOM. The last one is the programming interface of the HTML (Hyper Text Mark-up Language). HTML DOM defines the HTML elements as objects, properties for the elements, methods for accessing the elements and possible events for the corresponding HTML elements. (W3Schools Site 2017, The HTML DOM)

Browser interprets HTML to the DOM in-memory and lets the developer to insert, update and delete the DOM nodes / HTML elements or their properties with JavaScript and CSS (Cascading Style Sheets). JQuery framework made it easier for developers to manipulate the DOM compared to the basic vanilla JavaScript, but with even JQuery or equivalent frameworks, the DOM manipulations still had a poor performance. DOM was not made with performance in mind when considering dynamic UI manipulations. This means that when you try to change the order of several nodes or their properties, there are time-penalties associated with the operations that increase hand-in-hand with the amount of operations done. DOM is tree structured and you can use traversing techniques to find the DOM nodes up and down the tree within parent and child nodes. (Medium Site 2016, What is VirtualDOM)

Most of the applications these days are not static HTML pages, but instead they are dynamic pages, which have data that changes over time. Majority of the existing web pages are dynamic SPA’s (Single Page Applications). These applications hold their own State inside the application on the Client-side and calling the server may change the application State on the Client-side, but the management of the State is not dependent on it anymore. The paradigm of State handling has changed more toward the Client-side in web applications.
5.2 VirtualDOM

VirtualDOM has been introduced to the world before React but React made it to a familiar concept in the JavaScript community. Facebook made and open sourced its own VirtualDOM implementation, and by open sourcing made it available for all. VirtualDOM acts like an abstracted copy of the real DOM and is more like a blueprint architecture of the actual DOM. VirtualDOM’s abstraction gives an added value from the very beginning by being detached from browser specific implementations and thus being cross-browser applicable. Real DOM operations are avoided by the simplified copy of the HTML DOM. By being just a copy of the DOM, VirtualDOM can avoid slow and browser specific DOM operations that are costly in time consumption and performance. (ReactKunfu Site2015, The difference between VirtualDOM and DOM)

5.2.1 How the actual DOM works

The Browser engine that is responsible for rendering the requested HTML to the screen will go through sequential parts that are: Parsing of the HTML and constructing Document Object Model (DOM) from that. Construction of the Render tree and layout for it and finally rendering or painting the render tree on the screen. The event of parsing HTML document and its elements to Nodes in the DOM, will lead to a structure that is called a content tree. Style elements and CSS files will together formulate the render tree. The formulation of render tree will eventually lead from layout process to the actual rendering / painting stage. This sequential process will not wait for the whole HTML document to be parsed, instead it will incrementally render the parsed content on the screen.

Figure 1 Rendering the DOM
(Source: Taligarsiel Site 2017, Browser parsing general)
This process is done all over again every time the DOM changes. The parent element in the DOM and its children will go through sequential flow with each phase every time the data changes and re-render / repaint it to the UI. (Medium Site 2017, How VirtualDOM and Difffing works)

5.2.2 How the VirtualDOM works

When React element gets rendered, then whole of the VirtualDOM gets updated. VirtualDOM creates a snapshot of the previous state of the application before the update and diffs (differentiates) the new state that got updated to the snapshot state. By the diffing process React can figure out what has been changed and only update the changed parts to the real DOM by patching it. (Codeacademy Site 2017, React VirtualDOM)

Compared to a competing JavaScript dataflow such as Angular 1, which uses dirty-checking for in-order-to detect changes, diffing seems to be more sufficient. Model dirty-checking in Angular uses digest cycle, which loops through the Scope objects, which are watched for data changes. This comparison happens on each digest loop, which can be seen as a lag in the application. This happens, if the complexity of the Model grows upon watched elements. (Teropa Site 2013, Make your own Angular). Angular mutates the existing model to be aware of the changes and this becomes a problem, if the source of the data is new or an immutable data. The diffing in React is made on the visible VirtualDOM that is on the UI on each moment of time. This limits the available needed changes compared to Angular Scope objects. React VirtualDOM saves a lot of time with the diffing algorithm, because it only modifies / updates the DOM, when it's needed and with what is needed. DOM modifications themselves are notoriously time consuming, because CSS styles and corresponding layouts need to be calculated and applied with every modification of the DOM. The diffing algorithm is a lot faster and more efficient than the real DOM operations, partly because it only updates the subtrees that need to be updated after the changes occur over time. The only real disadvantage of the virtual DOM is the higher memory usage. The higher usage of memory comes, because of the full copy of the DOM that is needed in-memory. (Stackoverflow Site 2014, Why is React VDOM more performant than Dirty Model checking)
6 Building the VirtualDOM with ReactElements and user defined Components

With React’s declarative nature the developers can tell, which nodes need to be updated instead of manually traversing the DOM tree with every event, which is usually very error prone, hard to maintain and extend the code or to test it in anyway.

React VirtualDOM nodes can be divided into two categories and these are ReactElement and ReactComponent. ReactElements represent the real HTML tags, which are abstracted by React. "ReactElement is a light, stateless, immutable, virtual representation of a DOM Element" – (ReactKunfu Site2015, The difference between VirtualDOM and DOM). When ReactElements are rendered, then can the conversion to common HTML elements happen. ReactElements need to be immutable, because of the diffing algorithm. The diffing happens between the current and the new node. If the element would be mutable, then the change would just mutate itself to the VirtualDOM instead of the diffing and patching of the DOM with just the changed data. So, it is easy to justify the immutability here. ReactElements are just JavaScript objects, which have four properties. These properties are Key, Props, Type and Ref. Key is used with arrays to identify each unique element in the array. Props are the properties passed down to the child elements from the parent element and this is the immutable part of the ReactElement as you cannot modify them directly in the child elements. Type references to a valid HTML element. Ref is used...
for accessing the elements, which are already rendered to the DOM. (Medium Site 2016, React Elements vs. Components vs. backing instances)

The transpilation of ReactElement with references to the actual DOM node happens when JSX transpiles React.createElement to function calls and returns JavaScript objects from the function calls. ReactDOM then takes in the ReactElement and the reference to the DOM node and returns the instance of the element. This newly formed element is called as a Component Backing Instance with access to the DOM. Only ReactComponents with State will return a Component Backing Instance, as Components without State will return null. These backing instances are stored in-memory as objects. Components that have no State, are usually just function calls, called pure functions and they render the State of the view with changes on the props and they cannot be targeted with ref keyword, because they do not have Component backing instances. (JakeTrent Site 2016, React stateless components missing)

ReactComponents are where the State lives in React and they are mutable. ReactComponents are user defined classes, which inherit from React.Component as subclasses. ReactComponents have lifecycle events, a constructor (if they are defined in ES6 classes or higher instead of React.CreateElement functions) and a render method. ReactComponents do not have access to the DOM themselves, instead ReactComponents are converted to ReactElements in the render method of the instance and then these newly converted elements are patched to the VirtualDOM as Component Backing Instances for the differ algorithm to happen. (Medium Site 2016, React Elements vs. Components vs. backing instances; ReactKunfu Site 2015, The difference between VirtualDOM and DOM)
7 The M and C to complement the V

V in MVC keeps popping up everywhere when in contact with React documentation and it does not matter if the documentation is official or produced by a third party. What is the V or better yet, what is this MVC?

MVC is short for Model View Controller and it is an architectural pattern aiming for separation of concerns. Separation of concern means that it divides the application to three consistent logical domains, which handle their own domain. M is the Model of the data, a kind of blueprint of the data, which draws out the boundaries of logical this domain. V is how to present the Model data in a User Interface layout. C takes in the User inputs, manages the Model and updates the corresponding Views. MVC is all about the separation of concerns and initially it was used to allow multiple UI's for the Models in-order-to control the flow of the data. (TutorialsPoint Site 2017, MVC Framework introduction. Medium Site 2015, Flux vs. MVC)

React’s only concern is to present the data, as in being the View layer of applications. We can come to the conclusion that React is not a full blown mature JavaScript framework with full comprehensive solution of how to answer all of the needs in application development, but instead it is just another block or module in a paradigm that is modular JavaScript. Modular JavaScript in a higher scale is where you build your own framework by yourself by piecing it together piece by piece. React is its own ecosystem with various options of how, for smaller and bigger applications. (Toptal Site 2017, Navigating the React Ecosystem)

![MVC Pattern](Source: TutorialsPoint Site 2017, MVC Framework introduction)
The main idea behind React ecosystem is behind the unidirectional Flow of the Data also known as one-way data flow. Facebook left MVC kind two-way data binding behind, because that did not scale well for Facebook's needs. MVC introduced a lot of problems for Facebook in bidirectional binding of the data between Model and the View. This binding effect lead to cascading effects of data across the application, which in itself lead to poor maintainability, error prone code and difficulties in testability and reproduction of occurred errors in production. In a nutshell, it lead to more complexity in the data Models and it was often very difficult to see what was happening with the data flow. Simplicity was chosen as the driving force of React and unidirectional dataflow was chosen as the driver of React ecosystem. The data flows from top to down through React Components and React itself supports one-way dataflow but for larger applications you need something that supports the Flux pattern, which was Facebook's answer for solving the cascading effects and complexity of MVC patterns. (Medium Site 2015, Flux vs. MVC)

React components belong to a tree hierarchy where the data flows, from up to down, from the parent component to the child component. In an ideal scenario the parent component “owns” the State, which is distributed down as props (properties) to the child components by JSX attributes, which are XHTML like attributes in the defined element. The parent nor the child does not know where the State is held in the tree of components. Components are transformed to React elements, which form the Virtual DOM. The components are isolated from each other and own their own State. The components can be stateful or stateless and they cannot know, if the other components are stateful or stateless, because they are isolated in their own world. Any State change causes a re-render of React application and the diffing algorithm of what has changed begin from the top of the tree down the tree. (ReactJS Documentation 2017, Data flows down)

7.1 React component state

User defined React Component owns its local State. This is that for every defined Component there is a State, and nothing can access that Components State. You may pass props to the Component from outside of the Component in a Unidirectional way and you may pass props to the child Components from the Component in question that owns the child components in the render methods return statement, but you cannot access that Components own local State from anywhere else. Every Components State lives in isolation in a function scope. State is where modifications are made to the State, through user events or live cycle events on the Component. These modifications can be passed as props to the child Components. (Ubervu Github Page 2017, React guide – Props vs. State)
State can be set with `setState` function, which takes the initial state that is given to it and mutates / changes it by the given value. (ReactJS Documentation 2017, Component state; Codeburst Site 2017, React state vs. props explained) Props in the other hand are immutable and the parent element is responsible for the changes in props. This is what one-way dataflow is in React without any other state management tools. State and props both are responsible about the render methods output values and they are both in essence just plain old objects. (Codeburst Site 2017, React state vs. props explained)

There can be two types of available Components and these are Stateless Components often referred as Dumb Components and Stateful Components, which are often called smart Components. (Ubervu Github Page 2017, React guide – Props vs. State)

Stateless Components do not own any state and they are meant as purely presentational components that are basically immutable functions presenting the values given it by props. Stateless Components are meant for to be reusable as they do not own the state and have no way of knowing what they are used for. Data that they need are only given to them by passing on props to them, they do not have internal state or connect to external stores for data. (Ubervu Github Page 2017, React guide – Props vs. State)

Stateful Components on the other hand can access an external data storage, if needed or it can have its own internal state. React internal state is not needed to make a stateful / smart Component, the Component must change the state somehow or react to changes in the state. The Component can be a React internal State Component or it can connect to external state management tool following the Flux pattern or other tools such as Redux or MobX, which are the industry de-facto tools at the moment. (Preact GitBooks Site 2018, Smart Components) When stateless Components are focused on visual presentation, the stateful Components are responsible of networking between client and server side and they are responsible for reacting to the events on UI triggered by the users. (Ubervu Github Page 2017, React guide – Props vs. State)

Using React’s internal state may be problematic because it is local to the Components, and this is why it is called as local state. In large applications usually, local state will not suffice because of its nature of being local to the Component. Developers will run into problems when Components need to share state between themselves or they mutate the State of other Components. (RobinWieruch Site 2017, MobX Confusion) To share the state, you can use state hoisting, where you hoist the state up to make the state sharable
between Components. (Medium Site 2017, Hoisting state) This however becomes troublesome after a while when you cannot guarantee the limitation of bound state mutations to one Component only. (RobinWieruch Site 2017, MobX Confusion)

Michel Weststrate the creator of MobX identifies three kinds of problems with the React local state.

Firstly, setting the local State with setState is asynchronous. Setting the State directly by mutating it will cause the state being asynchronous. You should use setState as pure functions by returning new values of the State in an immutable way. This might do to keep your data up-to-date and thus avoiding the major problems of asynchronous state data. (Cloudboost Site 2016, Why I stopped using React setState; ReactJS Documentation 2018, React Component)

Secondly, setting the local State with setState will cause rendering to happen sometimes unnecessary. Calling setState will cause a rendering cycle to always happen. Sometimes this leads to unnecessary re-rendering of the Component (Cloudboost Site 2016, Why I stopped using React setState), if shouldComponentUpdate() is not implemented. Should-ComponentUpdate logic takes the previous and the new State and compares them, and if they aren’t different it will stop the Component from rendering, otherwise it will pass the State and cause re-rendering of the Component. (ReactJS Documentation 2018, React Component)

Thirdly, setting the local State setState will not do when working with more complex Components that need timers, networking etc. Often to manage these situations, the React lifecycle hooks are used with setState. Using the setState method in these lifecycle hooks with for example networking might lead up to very subtle bugs that are hard to test and repeat. (Cloudboost Site 2016, Why I stopped using React setState)

There are several reasons why to avoid React’s own local State but the main thing responsible for State management tools in React ecosystem is that it is near impossible or at the very least extremely difficult to handle State in complex applications with React’s own local State. This is why with the introduction of React composable Components and one-way data flow, the Flux architecture was not very far behind.
8 Flux an architecture to flow the data one way

The Flux homepage explains Flux like this: “It's more of a pattern rather than a formal framework” (Facebook Github Page 2017, Flux main page). It is a new kind of paradigm that is meant to deliver simplicity in the form of unidirectional data flow and complement React’s composable nature. Flux is just an architectural pattern meant to help you to better structure the applications, which are of the composable nature.

The unidirectional dataflow in Flux is happens through four interconnected definitions. Flux pattern definitions are: Action, Dispatcher, Store and View (Figure 4).

Actions are just simple objects that hold the Type property and the payload data attached to it. Actions in plural, concise the Application Interface (API) in the application. The type definition property in the Action object is the description of the interaction that will take place by user interaction or any asynchronous actions such as networking, db queries or timers. (Facebook GitHub Page 2017, Flux concepts)

The Dispatcher is usually a singleton in the application or it should be a singleton. Singletons are single instances of the object in question and provide a global access to the singleton in question. (Dofactory 2018, Singleton design pattern) Dispatcher has Stores registered to it and it receives all the Action objects dispatched to it and it dispatches these Action objects to every registered Store. All the Stores will get every dispatched Action through the Dispatcher. (Facebook GitHub Page 2017, Flux concepts)

There are several stores in Flux and these stores hold the applications logical domain, which in their turn hold the applications whole State. View listens to the Store that is associated to it and re-renders whenever the listener notices a change happened in the Store. Every time the store receives an Action through the Dispatcher that it is subscribed to, it checks the Action type and mutates the state by responding to the Action type and payload data and then it uses its event emitter to emit the state change to the corresponding view. (Facebook GitHub Page 2017, Flux concepts)
The flow of the data is under strict rules and is very opinionated when using the Flux pattern. Changes defined in Actions must go through the Dispatcher and to the multiple Stores, which live in isolation from each other. (Facebook GitHub Page 2017, Flux concepts)

8.1 Flux implementations

The problem with react State is actualized in moderate or large size applications pretty quickly. Even though, you can access the State locally with setting and reading the local Component State, this will become cumbersome or chaotic with larger applications that have components that share State. In using just React State components that share State with other components or components that mutate the State of other components need to be placed inside of a tree like component hierarchy. Depending on the State changes you need to either hoist your components up or down depending on the kind of State changes you need. This leads to messy code that usually is not so transparent in functionality. This also may lead to the possibility of that you’re changes to the State may leak to components that they were not meant. This may be one of the reasons that lead to the development of Flux pattern and derivative State containers such as Redux or MobX. (RobinWieruch Site 2017, MobX Confusion)

There are several Flux implementations that are available these days and some of them expand and modify the Flux architecture in their own way. Flux by its own just an architectural pattern, which was designed by Facebook to offer ways to manage the State of the application. This pattern was meant to be handed out as a reference to the JavaScript opensource community and was meant to be a very opinionated guideline for best practices concerning State management. The real implementations of libraries or frameworks were left to the community of developers. Facebook kind of pushed Flux pattern to the world, and thought that by time, the problem of optimizing both performance and complexity of Flux, would be settled by the community, that would tackle these issues.

The complexity of Flux implementations came with the amount of boilerplate needed to follow the paradigm of Flux architecture. The users / developers using Flux found out that a lot of the responsibility of implementing Flux was left to the imagination of developers, who found out that they were stuck in most cases when it came to asynchronous data flow for example.

The first true Flux implementation, and alternative mature library that caught on, was called Reflux. It merged the main concept of Flux dispatcher with action creators (func-
tions) and actions (objects). Dispatchers were multiplied from one to many and this simplified the maintainability of the code through the isolation of the dispatchers. This isolation however came at the cost of debuggability of the code and made it harder to maintain asynchronous code. (JamesKNelson Site 2015. Which Flux implementation should I use with React)
9 Redux to predict the state in an opinionated way

Redux is a Flux like library, introduced to the JavaScript community by Dan Abramov. with a small footprint of 2kb in total size (ReduxJS GitHub Page 2018, Main page), Redux follows the Flux pattern but makes additional arguments to the paradigm and in the way, it simplifies and solves a lot of Flux problems. The problem of Flux is often multiplied by many Flux libraries doing the same thing in various ways that have a lot of inconsistencies between them, and this has led to deep confusion of how to do things among the users / developers using Flux implementations. Redux in a way, is Flux morphed and matured in to something, that is a standard nowadays in the React community.

Dan Abramov described the evolution of Flux in his blog back in 2015 and stated that the State should not be owned by the store. Here, he was on the verge of making Redux and described the changes to the Flux paradigm in a few ways. Here, he expressed that the Stores and Actions (Action Creators) should be expressed as pure functions. Later, when Redux marched as shipment ready, the Stores in plural were reduced in to just one Store as the source of the State and Reducers handling the State changes. Statefull Stores were to change to just one Stateless Store and thereby, instead of mutation there were immutability. (JamesKNelson Site 2015. Which Flux implementation should I use with React; Medium Site 2015, The evolution of Flux Frameworks)

The crucial difference between other Flux implementations and Redux, is that Redux implements a single Store instead of multiple Stores and avoids State mutations in all cases. Redux dumps the Flux dispatcher required in Flux and instead of the dispatcher it uses plain objects or action creator functions in the stores dispatch functions. (SmashingMagazine Site 2016, Introduction to Redux) Flux on the other hand uses event emitters that broadcast changes by type and payload data to the subscribed components as call-backs. Instead of these event emitters, Redux uses easy composable pure functions that always produce new objects in an immutable way instead of mutating the State variables directly when changes occur. (ReduxJS Documentation Site 2018, Prior art)

9.1 Why Redux

Redux was the love child experimentation of Dan Abramov gone wild. The initial goal of D. Abramov was to produce better developer tools for the Flux pattern, but in the way the project divided in to an altogether different pattern than Flux. Abramov wanted to implement Time-travel and hot-module-reloading (HMR), but these turned out to be cumbersome to implement following just the traditional Flux pattern.
HMR means that the state is kept in development time during the changes in the code. When changes are made to the code, HMR makes it possible to keep the state of the application, even after the page refreshes. With the use of HMR you cut the repetitive user input of the same State data during development time. (CodeCartoons Site 2015, HMR and Time-Travelling)

Time-travel is the work of magic made possible by immutability. Time-travel makes it possible to “travel through time” of State change events. This travel from and between State changes improve debuggability and therefore it improves testability of the applications even with error events in production environments. (CodeCartoons Site 2015, HMR and Time-Travelling)

The main reasons why Time-travel and HMR were difficult to implement with following the traditional Flux pattern were:

1. The Flux store shares the State and the logic to change the State in the same exact place. Without the separation of these two, it made it near impossible to change the code without destroying the current State and this was the main problem that affected HMR. (CodeCartoons Site 2015, Intro to Redux)
2. Mutation of the State made Time-travelling impossible. As the State gets mutated, it gets overwritten with the new data and one State object with pointers. One object and pointers to it, meant of not knowing, what was the State before. (CodeCartoons Site 2015, Intro to Redux)

9.2 Redux loves Functional Programming

Redux is built on top of the concepts of Functional programming and it is essential of understand a few pointers of it and how it differentiates from imperative programming. As imperative programming use is a lot of times pointing to OOP (Objective-oriented Programming) and to its patter that leans heavily in the use of interfaces, classes and the methods that they expose from within. Functional Programming takes another way. FP the abbreviation of Functional Programming treats functions as a first-class citizen and this fits JavaScript perfectly as it does the same as a language with multiparadigm. (Medium Site 2017, What is functional programming)

Imperative programming can be understood for what it is, and this is the word commanding. The flow of the code must be as is, synchronously step-by-step to produce what is wanted. Imperative is the How in doing things. How to do things is the flow of the code. (Medium Site 2017, What is functional programming)
The synonym for declarative is explanatory and it points to *What* instead of *How*. Declarative is FP. Functional Programming leans on expressions, which are used on value evaluations. Shared application State is to be avoided through pure functions that withhold no side-effects and produce the same output given the same parameters always. The given parameters are not to be changed by mutations and you must return a new value instead of returning mutated value. Immutable objects must stay the same after creation, as they must not be changed (mutated). OOP and Imperative programming takes mutation to the core by shared State, where object or values in memory exists in a scope that is shared, and the values of that object can be mutated / changed directly when needed. The signature characteristics of FP is to use pure functions that may be composable, avoid sharing of the State and the mutation of that State in any means. The avoidance of side-effects is imperative (pun intended). Side-effects include any I/O (Input/Output) that is not predictable and may produce a different value each time, through time. Networking, such as asynchronous API calls, database queries and logging are examples of side-effects that makes a function impure. (Medium Site 2017, What is functional programming)

### 9.3 Redux state tree

“State is a plain JavaScript object. You never mutate it directly, but instead, derive a new updated state when something happens in the application (which result in action dispatch) through pure functions.” (Stackoverflow Site 2017, Advantages and disadvantages of MobX and Redux)

Flux signature pattern can be thought as \( (\text{state}, \text{action}) \mapsto \text{state} \), but even though Redux pattern can be interpreted to be the same, it is still more like \( (\text{oldState}, \text{action}) \mapsto \text{newState} \). State is a tree like structure, often referenced as the single source of truth in a religious like dogma. This State-tree in the end of the day is a simple JavaScript object consisting of maps and arrays of pure data. State-tree is immutable, and you cannot change its data in the trees live span, instead you use the tree to derive the seed to spawn a new tree from it. This new tree reflects upon the changes made to the tree with the pure new data. (Teropa Site 2015, FullStack Redux tutorial)

Action is dispatched with the oldState State-tree object and the action itself holds the type of the action function dispatched with the fresh payload data. The oldState with the Action object is put through a reducer, where the newState is reduced / deducted from. Every change dispatched with Action object data is reduced to a completely new tree. This tree can be treated as a snapshot of the State history, which is serializable. With this snapshot you get the current and previous State-trees in-memory and it is quite easy to reason about undo / redo functionality after this. Serializable State with previous snapshots make
it possible to send the State through a network for later evaluations on error situations. (Teropa Site 2015, FullStack Redux tutorial)

9.4 Redux API core concepts

Redux has a small library with a footprint of just 2kb. It is more like an opinionated paradigm, a pattern (if you will) to manage the State in the Client-side of web applications. The core concepts of what the Redux gives you as a library boils down to three things. These things are: The Store, Reducer and Actions. (ReduxJS GitHub Page 2018, Main page)

The only way to modify the State in Redux is through the use of Actions. Actions are only simple objects with one mandatory and second optional field. First one is the type field that is usually a string constant and the optional payload data. The type of the action (the mandatory field) us needed to tell the reducer, which part of the State is needed to update with the action payload data. (ReduxJS Documentation Site 2018, Actions) There are several situations where you can imagine that the action payload data is not needed and therefore voluntary, for example the State data could be a simple Boolean that will update only through the type of the action. If the type of the action does not match the reducers switch statement clauses, then nothing is changed, and the old State is returned. You can return a simple object through actions, but it is preferred to create actions through action creators, which are usually pure functions that are dispatched to the store by the stores dispatch method. (SmashingMagazine Site 2016, Introduction to Redux SmashingMagazine Site 2016, Introduction to Redux; ReduxJS Documentation Site 2018, Actions)

The name for Reducer came from the JavaScript array method reduce. In a nutshell reduce method can be explained in the following way: a single outcome value is reduced from multiple values from arrays or object fields. In Redux the multitude of values are functions that are actions with type definitions, that correspond to the Reducers switch statement clauses. These clauses are meant to change the slice of the State that the Reducer in question is responsible off. (SmashingMagazine Site 2016, Introduction to Redux)

The Reducer parameter signature is \((state = initialState, action) => newState\). This means that the parameters are the State that is the applications State before any modifications and the action object. Actions are given to the Reducer with store dispatch method. Action object has the type definition and payload data (as described above) and these are given to the Reducer. Action object with the type definition corresponds to the switch statement clause that triggers the function to change the state in a desired way. The Reducer gives the description of how the State changes. The Reducers functions may not mutate the
data of the State as they must be pure functions or return pure functions. For the functions to stay pure they need to avoid side-effects, such as networking API or DB requests or other non-pure functions in general. Stay pure and calculate. Calculation here means that given the State before and description of what to change in the State how, calculate a new State from this. (ReduxJS Documentation Site 2018, Reducers; SmashingMagazine Site 2016, Introduction to Redux)

There is only a one store defined in Redux. The Store gives you access to the State that is just a simple object (State-tree) that must stay in a serializable form (easy JSON conversion) and it is better to normalize it avoid data duplication in deeply nested data. Think about database normalization, where id’s for example are used for entity parent / child relationships. The Store exposes a few helper functions, such as the dispatch function, which you must use to produce a State change in Redux. (ReduxJS Documentation Site 2018, Glossary; ReduxJS Documentation Site 2018, Store)

9.5 Async Redux

Redux works as synchronized step-by-step by default. Redux does not have an opinion of how asynchronisity is supposed to be baked in. Dan Abramov, the creator of Redux states that Redux is a low-level tool for State management and adds that developers can use external libraries for async in Redux or built supersets on top of Redux themselves. (Stackoverflow Site 2016, How to dispatch a function with a timeout)

Syncronous flow of data, step-by-step is what Redux supports by default. Middleware is a place that you might add customizable or third-party functionality to your application. Middleware is a wrapper function for the dispatch function of the store. It intercepts the dispatch call to the store, so you can add asynchronous actions alongside normal ones. Composition of multiple middlewares makes it possible to chain middleware libraries together. By middleware you might think of a procedure that intercepts the default behavior of a function to add something on top of the original function with middleware function. In Redux it is meant as a point of extension between action that has been dispatched and the Reducer where it has been dispatched to. (ReduxJS Documentation Site 2018, Apply Middleware)

This means that actions, reducers or the store themselves are not meant to solve the problems of asynchronisity, Redux gives only its opinion of where to fire async actions and this is the action creators. On low level with less complex applications this might do well, but the boilerplate and handling of the code structure grows immensely difficult to
maintain with more complex applications. This is where Dan Abramov suggest using outside help in the form of a library that solves this problem efficiently and adds a clear-cut architecture on how to do async in Redux. With the help of such libraries, such as redux-saga or redux-loop you can solve this problem, but with the addon of complexity. Redux-thunk is a middleware library that solve this eloquently with small case applications and it suffices for them, but with anything more complex you will need a library such as redux-saga. (Stackoverflow Site 2016, How to dispatch a function with a timeout; ReduxJS Documentation Site 2018, Async flow)
10 MobX for less complexity

Using Flux pattern like Stores to update the values due to user events or incoming events by components subscribing to changes is cumbersome in the long run with managing the subscriptions. Every subscription that is informed about the changes, will make a recompute to the components by these subscriptions. The problem here, is that this subscription to the component or components are left to the developers themselves. Problem that leads often to oversubscribing and therefore more computations that is needed or under-subscribing and therefore stale out of date data. MobX trade-off is in-memory usage instead of CPU cycles, which is cheaper in single threaded JavaScript runtime environment. (HackerNoon Site 2015, Becoming fully reactive)

10.1 Avoid staleness through Reactivity

MobX creator Michel Westrate states that pure rendering through time and State was the inspiration behind MobX in his blog back in 2015. The application State management systems mainly back then and now morph the UI (User Interface) View to State and vice versa. Each change in the State can lead to stale data. This is due to the programmers immerse responsibility of keeping up with the subscriptions between Views and the Store / Stores. Subscriptions are watchers / event handlers that watch the changes to the data and take care of updating the view with up-to-date data when they are notified with changes to the corresponding data. This type of pattern with subscriptions will lead eventually to the addition of overhead boilerplate code that is often error prone and adds to development time, oversubscriptions that hinders performance in the form of CPU cycles.

The idea behind of MobX is not to subscribe / unsubscribe to the State, instead it uses isomorphism / connectivism behind the values in the view and the values in the State. FRP (Functional Reactive Programming) does this with streams and it is implemented in such JavaScript libraries, such as Reactors BaconJS and RxJS. MobX does this a little differently and adds a letter to the FRP paradigm. TFRP (Transparent Functional Reactive Programming) creates an isomorphic relation between data and events through expressions and data graphs the same way that for example excel spreadsheets does. This makes it possible to produce atomic changes in the data that does not render any intermediate results in the View. In spreadsheets one data cell can be the result of a computation between two or several other cells. This refers to the idea of pure rendering with pure functions that do not produce any side-effects and therefore does not mutate values. Pure rendering in MobX refers to the computed values, which are the sum of observables.

Computed values are the sum as observables are the values that produce the sum. (Medium Site 2015, Pure rendering in the light of time and state)
MobX leans heavily on the TFRP paradigm, which is an abbreviation to Trasparent Functional Reactive Programming. The concept of FRP can be derived from just two things, which are Observables and Observers. Observables form the State in a single source of truth as the State of the application. This source is then observed by the Observers, which derive the values from it and update it accordingly. (HackerNoon Site 2015, Becoming fully reactive)

MobX hammers down State management in such a way that it abstracts it behind a curtain and automagically handles and produces a consistent up to date State for you. You can get lost in the way of implementing a consistent State with State engines / containers such as Redux. Redux itself makes the State immutable and with it loosely opinionated style forces Functional Programming style and boilerplate behemoth type of code to the developers. This introduced boilerplate coding style to several different files and added complexity, which is a mandatory and tedious procedure in every Redux project. With Redux, you have to follow functional programming style and understand the pitfalls of it. With Redux you need the data to be normalized and referential integrity is left as a question mark. (HackerNoon Site 2015, Becoming fully reactive) With Redux you cannot benefit from the multiparadigm of JavaScript as you must avoid the imperative style OOP concepts, which you can implement in JavaScript as Prototyping. JavaScript is dynamic, multiparadigm language and managing the State should be that as well. You choose the style and not vice versa. This is the power of MobX and the downfall of it as well. MobX does not have the opinion of How, it has the opinion of What, as what you get with MobX and what you get is freedom.

Stale State with inconsistent values is what you can avoid with MobX. Consistency is guaranteed through derivations from your observed values / observables that get automatically updated when values / observables update, and this means consistent State. State withholds the complex graphs of data, computed values that compute new values from the observed values, actions that mutate the State values / observables and finally reactions that do not themselves produce any new values but react to the State values changing. (HackerNoon Site 2015, Becoming fully reactive)

10.2 The gist of three behind MobX

Basically, MobX boils to just four things, which are to be understood in for to understand MobX. These are observables, computed values, actions and finally reactions. The third and fourth are themselves equivalent of derivation concept in MobX.
JavaScript class is either decorated with the @observable decorator or either with the observable function with declaration without class definitions to make data observable. These observables are either complex data graphs containing objects, arrays or references to other observable objects or just JavaScript primitive values. The conceptual Derivation is factually the computed value or values in MobX. Computed values are values that are computed through pure functions with no side-effects, such as networking, database connections or any other means that may mutate the State in any way. Actions are the quintessential place for just mutations as they’re only means of existence is to produce side-effects by mutating the existing observables and State. Actions are in fact the preferred way of mutating the State just to keep the referential integrity intact and by this to avoid “state soup”, which was introduced in away by two-way binding libraries such as the first Angular version library’s. Two-way binding like functionality can be achieved by mutating the observable values without actions by turning off MobX Strict-mode. This though is not the recommended way of using MobX as you get one-way data flow through mutation through actions and thus avoid the treacherous and confusing “state soup”. Re-actions are derivations from observable data but the similarity to computed values end there in such way that reactions do not themselves produce any new values themselves. Re-actions holds they’re place in situations where you need I/O relative tasks, such as networking or update the DOM. MobX inner workings make sure that all that can be derived from the State, will be derived such as computed values or reactions. (MobXJS Documentation Site 2018, Overview; MobXJS Documentation Site 2018, Getting started)

### 10.2.1 The Observer

Observers acts as a bridge between React components and MobX handled State. This is a binding that is brought from mobx-react package. Observer makes all wrapped functions reactive for observing the data within. (MobXJS Documentation Site 2018, Overview)

Observers react to the observables data change in components, be they smart or dumb and are to be applied in components that need to render observable data. Subscription problems that affect usually Flux implementations are avoided by Observers subscribe to data that was involved in the last or latest render. The determination of dependencies happens at runtime in a matter that is fine-grained and the Reactive way of loading the data prevents unnecessary renders of parent elements when only their child elements data has changed. (MobXJS Documentation Site 2018, Observer)
10.2.2 Explicitly actions

Actions are needed when the code needs to change the State. Such situations include, such as events triggered by the user, schedulers, websocket data pushes from the server, networking. Basically, any kind of situation that produce side effects are ideal of using an Action in MobX. (MobXJS Documentation Site 2018, Concepts)

With using Actions, you get several benefits, which include achieving explicit one-way dataflow that runs synchronously and you get better structuring of the code in the same time. (MobXJS Documentation Site 2018, Concepts)

It is considered as best practice in the word of MobX to use Actions only and explicitly modify the State through them, instead of mutating the values implicitly by assigning values straightly to the observables. MobX does not come with Strict-mode enabled out of the box and as a developer you must enforce this yourself. Mutating values in Strict-mode only through Actions comes with the benefit of optimizations as it applies wrapping of Action methods / functions in transactions. Transactions run mutations in batches, synchronously in order. This means that when the last action that is inside the transaction batch will finish, the derivations (computed values / reactions) will be notified by the State mutations. By running mutations synchronously, MobX avoids producing intermediate values and therefore a stale State. (MobXJS Documentation Site 2018, Action; MobXJS Documentation Site 2018, Overview)

10.2.3 Derive all

MobX has two kinds of derivations and these are Computed values and Reactions. Derivation means a value that has been derived from the State or computed value derivations without outside influence automatically. Computed values are pure functions without side-effects that are derived from the State observable values or other computed values. Computed values have the same concept that formulas in Excel or other spreadsheets. MobX core concept in using computed values is to keep you’re State as small as possible as in observables and derive from those observables as much as you can in the form of computed values. Computed values have lazy evaluations and they are called only when needed, for example then when the view is in use. When the Computed value is out of focus, it will be automatically garbage collected. Optimizations also have the effect in the form of caching, that when computed value is a pure function and the parameter values of that function do not change then the function won’t run in the first place. (MobXJS Documentation Site 2018, Computed decorator)
Reactions are derivations that are used to produce side-effects in a reaction to changes in the observable values. Derivations get the notifications in the changes in the State in a synchronous process and they will be processed after the notification automatically and in between there will be no stale values to them. Perfect example of a reaction derivation in MobX, is the libraries autorun function. Unlike computed values, autorun reaction won’t be garbage collected when the function is run. You must close the side-effects your ‘self.

(MobXJS Documentation Site 2018, Concepts; MobXJS Documentation Site 2018, Computed decorator)
11 Project Research plan

11.1 Background

The writer of this thesis has worked now on the time of writing this specific line, three years as a fullstack developer in a Software Consultancy Firm named HiQ Finland. And in that time the writer has been a part of several projects ranging from big to small. This thesis has its roots on the latter part. A proof-of-concept application and a sneak peek to the case of MobX-state-tree through it. What does MobX-state-tree bring to the table in the battle for betterment, concerning the case of State management machines in the React Ecosystem.

The writer of this thesis started this thesis almost two years ago, in a time when there were a lot of confusion about React and how to manage the State within it. As ever the industry takes a time of relaxation upon technologies adopted to the status of de-facto, such as AngularJS was just a few ago. The time of AngularJS passed as the new kid on the block, namely React came to the spotlight and stole the thunder of the weary JavaScript framework. On the emergence of React came a new kind of thinking, that made modular and unidirectional dataflow popular, but this lead to a lot of confusion, of how to do it properly and which tool to choose on the handling of application State. There were suddenly not just one framework API as the backing instance to fall upon.

The theoretical background of this thesis handles the emergence of ReactJS in chapters that are structured in a chronologically like of a way. Chronological way, that spans from the times of old when JS was developed, to the emergence of competing ways of holding State in a React application. This thesis study lies on the foundations of old with something new to bind them together. The curious case of mobx-state-tree as the headline of a blog by Michel Weststrate states all that is needed in this prologue.

11.2 Research Scope and Objectives

The writer of this thesis was assigned as a lead FullStack developer to a project that was small enough in its stature to be developed by a single developer. The architecture, visual design and the actual coding was assigned to yours truly and done by yours truly. I chose to use this application in my study as I decided to use MobX-state-tree as a main component of the application.
Application itself is vailed upon secrecy concerning the server-side code with a non-disclosure agreement made with the Customer. Despite the fact, I can disclose some facts upon the server-side, which won't break the agreement, but let's just say that the server-side is out of scope for this study.

What is in the scope of this study, is the client-side and its implementation on using MobX-state-tree. The target is to study how the implementation will take use of the library and how it complements the use of React and is it a possible substitute / replacement candidate for accomplished libraries, such as Redux or MobX?

11.3 Work described

The application was done in following a loose agile methodology with a deadline in sight. Discussions were held with the Customer in meetings, in order to map out the requirements for the application and to carry out the full architecture from start to finish.

Use cases, User stories and layout framework were mapped and designed in the first iteration before the start of application development. This thesis will show the stripped-down version of those. These versions were stripped down from original ones, because of confidentiality issues that those versions would hold.

The second iteration involved the implementation of the application in its static appearance layout on the Client.

The third iteration involved the mapping of the REST API server-side application for the persistence of the data in a way that is not to be disclosed by this thesis, because of confidentiality and the scope of this thesis. This iteration involved the dynamic management of the API calls and the state management of the Client application UI events as well.

Fourth iteration involved the test and production deployments of the application in the final environments, which I cannot discuss in detail, unfortunately.
12 Design

12.1 User stories

12.1.1 User story for the Administrator

As an Administrator I want to login,
As an Administrator I want to navigate between History, Command and Fast Command Screens,
As an Administrator I want to be able to send Fast Command of a certain specified type to one specific usage place,
As an Administrator I want to set the type of Command request,
As an Administrator I want to input several usage places to the Command and get the addresses for those usage places before I send the Command request,
As an Administrator I want to see the progress / state of the request,
As an Administrator I want to see if the request ended up in a failure,
As an Administrator I want to see if the request was successful,
As an Administrator I want to be notified, if there were any unsuccessful results in the request patch and identify the places of the failed places in the details of that request,
As an Administrator I want to see the details of successful items in the request patch,
As an Administrator I want to send several Commands, one after another and see their progress and results in a queue,
As an Administrator I want to be able to send several Fast Commands, one after another and see their progress and result in a queue,
As an Administrator I want to be able to use History search capability and filter the History search by several filters,
As an Administrator I want to be able to access the results of the History search in a table and to be able to navigate to the details of an individual Item

12.1.2 User story for the Electrician

As an Electrician I want to login,
As an Electrician I want to navigate between History and Fast Command Screens,
As an Electrician I want to be able to send Fast Command of a certain specified type to one specific usage place,
As an Electrician I want to see the progress / state of the request,
As an Electrician I want to see if the request ended up in a failure,
As an Electrician I want to see if the request was successful,
As an Electrician I want to see the details of successful item in the request,
As an Electrician I want to be able to send several Fast Commands one after another and see their progress and result in a queue,
As an Electrician I want to be able to use History search capability and filter the History search by several filters,
As an Electrician I want to be able to access the results of the History search in a table and to be able to navigate to the details of an individual Item

12.2 Use Cases

12.2.1 Use Case Login

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Login</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brief description</td>
<td>To be able to login to the application user must exist in the company X’s Active Directory in the Users Groups that are allowed to login to the Application service</td>
</tr>
<tr>
<td>Actors</td>
<td>Administrator, Electrician</td>
</tr>
<tr>
<td>Pre-Conditions</td>
<td>User must be found in the company X’s Active Directory in the Groups that are allowed access to the application. These Groups are either Administrator for the Application or Electrician.</td>
</tr>
<tr>
<td>Goal</td>
<td>Allow access to the authenticated and authorized users</td>
</tr>
<tr>
<td>Basic flow</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Authorized user, which can be found in the Active Directory Groups logs in to the service by AD username and password.</td>
</tr>
<tr>
<td>2</td>
<td>If the user is found in the AD and the defined AD Groups, then the user will get authorized and the user is permitted to login to the application service.</td>
</tr>
<tr>
<td>3</td>
<td>Permission and authorization towards the application service will be defined upon the user’s roles in the AD Groups. Administrator can send all kinds of Command requests. Electrician on the other hand can only send Fast Commands.</td>
</tr>
<tr>
<td>Exception Flow</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>If the user is not found in the AD or is not a member of the defined AD groups, the user is not authorized to use the service and user’s login will be prevented.</td>
</tr>
<tr>
<td>Final result</td>
<td></td>
</tr>
</tbody>
</table>
User is logged in to the server and the login credentials define the level of authorization of the user. The level of authorization defines, if the user is in the role of Administrator or Electrician.

### 12.2.2 Use Case for Fast Command

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Fast Command request type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brief description</td>
<td>Authorized logged in user in the Roles of either Administrator or Electrician can send request queries to determine the Current Voltage of a single usage place.</td>
</tr>
<tr>
<td>Actors</td>
<td>Administrator, Electrician</td>
</tr>
<tr>
<td>Pre-Conditions</td>
<td>User must be found in the company X’s Active Directory in the Groups that are allowed access to the application. These Groups are either the Administrator for the Application or the Electrician.</td>
</tr>
<tr>
<td>Goal</td>
<td>Allow sending of Fast Command request queries for authorized users.</td>
</tr>
</tbody>
</table>

**Basic flow**

1. Authorized user, which is found in the AD and in the defined User Groups in the AD can send Fast Command of the type of Current Voltage to a single usage place.
2. The send request shifts to the top of the queue to await upon a response from the server.
3. If the response returns as successful, the response shows the success mark on the corresponding request and the details of the response upon clicking on the request in the queue.

**Exception Flow**

1. If the request response returns as unsuccessful, then the failure will be shown in the correspondent item. The request may not pass the validation of the request body on the server side and an error message is shown on the Client.
2. If the server response status is either 401 or 403 meaning that the user is not authenticated or authorized to send the request, the user will be logged out automatically.

**Final result**

Authorized user in the roles of either Administrator or Electrician sends a Fast Command successfully.
### 12.2.3 Use Case for Command Request with a defined type

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Command request with a defined type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brief description</td>
<td>Authorized logged in user in the Role of Administrator can send a single Command that can be of various types to several properties at once.</td>
</tr>
<tr>
<td>Actors</td>
<td>Administrator</td>
</tr>
<tr>
<td>Pre-Conditions</td>
<td>User must be found in the company X’s Active Directory in the Administrator Group.</td>
</tr>
<tr>
<td>Goal</td>
<td>Allow sending of Command request query that may be one of several types for authorized users that are in the role of the Administrator. The View of the Command Form will be shown only for Administrator Users.</td>
</tr>
</tbody>
</table>

#### Basic flow

1. Logged in user, which has been authorized to the role of Administrator can send a Command, which can be one of various types to several usage places in a single request.

2. When user adds usage places to the list of usage places that belong to the Command request, an address belonging to the usage place will be pre-fetched from the server as a separate request. If the usage place address is found, then it is shown in the list next to the usage place name. If there are no addresses found belonging to the usage place, then unknown place label will be shown next to the usage place. The number of usage places you can add to the Command request is limited to 50.

3. When the Command is send with the type and usage places belonging to the request, the request will be placed at the top of the queue as a new Command request awaiting respond from the server. The send request will show the number of properties / usage place’s belonging to the request, the type of the request and the time when the request was send.

4. The request will poll the server recursively with 10 second intervals depending on the server response. The server status response can be:
a. 200, if the status of the request is successful but the procedure is not finished yet. This response status will determine the recursive request polling.

b. 201, if the procedure has finished successfully, the response body from the server will include two lists that are: successfull_commands list that include all the usage place’s, which were processed successfully with the given Command, and failed_commands list that include all the usage place’s that failed to process.

c. If the request fails altogether, the usage place’s send in the request will be shown as a failed_commands list.

### Exception Flow

<table>
<thead>
<tr>
<th>Exception Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> User in the role of Administrator sends a Command request that fails in the server validation step and all of the usage place’s will be added to the failed_commands list and an error message is shown on the Client.</td>
</tr>
<tr>
<td><strong>2</strong> If the server response status is either 401 or 403 meaning that the user is not authenticated or authorized to send the request, the user will be logged out automatically.</td>
</tr>
</tbody>
</table>

### Final result

Authorized user in the role of an Administrator send a Command with a distinct type to one or several usage place’s successfully.

### 12.2.4 Use Case search and filter history requests

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Search and filter request history</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brief description</strong></td>
<td>Authorized logged in user in the Roles of either Administrator or Electrician can use history search functionality with filters to search own or others send requests.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>Administrator, Electrician</td>
</tr>
<tr>
<td><strong>Pre-Conditions</strong></td>
<td>User must be found in the company X’s Active Directory in the Groups that are allowed access to the application. These Groups are either Administrator for the Application or Electrician.</td>
</tr>
<tr>
<td><strong>Goal</strong></td>
<td>Allow history search functionality for users that are authorized to access the application.</td>
</tr>
<tr>
<td>Basic flow</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>Logged in user, which is authorized in the role of either Administrator or Electrician can make a history filter request.</td>
</tr>
</tbody>
</table>
| 2 | The filters for history search include:  
   a. Usage places: User can determine one or several usage places to filter upon. Default filter is all usage places. User will be able to remove the added usage places.  
   b. Commands: add or remove Commands. Default filter is all Commands.  
   c. Request State: add or remove all the desired states (successful, error, pending) one-by-one. Default filter is all Request States.  
   d. User: add or remove all the desired users. Default filter is all users.  
   e. Starting date: add or remove the starting date of the send Command request. Default starting date is the beginning of the day corresponding history search query (00.00).  
   f. Ending date: add or remove the ending date of the send Command request. Default ending date is the current moment of the history search query. |
| 3 | History search query is send with the determined filters and request / response cycle is started. Loading screen is shown after the request as it is pending. |
| 4 | Successful request redirects to a screen showing two buttons, which are show history and new search. The amount of history results is also shown. History search is limited to the last five years upon the current request moment and the items in the response, which may be at maximum 10000 items. |
| 5 | Show results button will redirect the user to the paginated page including all of the results in the history search request. The page is paginated by 50 items per page. |
| 6 | Upon a single history search item result, the user can navigate to the details about that correspondent item. |

<table>
<thead>
<tr>
<th>Exception Flow</th>
<th></th>
</tr>
</thead>
</table>

41
1 User in the role of Administrator sends a Command request that fails in the server validation step and all of the usage place’s will be added to the failed_commands list and an error message is shown on the Client.

2 If the server response status is either 401 or 403 meaning that the user is not authenticated or authorized to send the request, the user will be logged out automatically.

3 If the server has responded with a 500-error status, the user will be shown a new search button.

**Final result**

Authorized user in the role of an Administrator or Electrician may send history request query by determined filters upon the request.

### 12.3 Layout wireframe

Figure 5: The essence of the application wireframe
Figure 6: Menu layout with logo placement

Figure 7: Command Form initial state
Figure 8: Command Form, secondary state before sending and usage places inputted by the user.

Figure 9: Queue pending state for three example requests.
Figure 10: History Filter form

Figure 11: History search result table
13 Application build with MobX-state-tree

13.1 Development Tools

The application that was developed for the Customer was decided to be a FullStack JS environment, meaning that the backend and frontend code bases share the same coding language, which is JavaScript. Thus, the choosing of main development tool was obvious and the choice landed upon Visual Studio Code text editor. Visual Studio Code surpasses its rivals (in thesis writers opinion) with tooling for JavaScript language it offers, such as debuggers and intellisense. VS Code also offers a wide array of extension plugins for the language, its supersets and libraries.

13.2 Application stack

13.2.1 Backend stack

The application developed was an NodeJS application using IISNode integration, which uses IIS (Internet Information Service) as a proxy server for hosting and scaling the Node process into multiple threats (Tomasz Janczuk Site 2011, Hosting NodeJS applications in IIS). Specifically, the developed backend was an REST Api with ExpressJS. The application was connected to the Clients Active Directory for authorizing every request to the server. This is just high overall description of the backend and I left the persistence of data out of the scope by purpose and this is just on need to know basis from this on.

13.2.2 Frontend stack

The Client SPA (Single Page Application) key components include:

1. React for the View Components
2. Styled-Componentss (CSS in JS)
3. MobX and MobX-state-tree for the management of the State and building of models
4. React-Router for routing on the Client-side (SPA)

React and MobX have been covered in this thesis earlier on and MobX-state-tree will be covered in this thesis study part. The two left main components that are in the need of some introduction / explanation, are Styled-Components and React-Router.
React-Router version 4.0 was used in the project. First before we can go to React-Router and what it is, we need to understand the need for a Router in SPA in the first place. Traditional web application routing comes with a request to the server and the server response includes the new HTML corresponding the navigation request. In SPA the routing does not work like this. The navigation needs to respond to the users request without a request to the server. The routers in SPA work on top of the JavaScript API window.location object and its functionality. URL (Uniform Resource Locator) is divided into five parts and these parts include:

1. The protocol used. (Either HTTP for insecure connection or HTTPS for secured connection)
2. Hostname (The host server DNS, which stands for Domain Name Server)
3. Pathname (The path of the page in application in the given moment)
4. Search (query parameters)
5. Hash (Setter or Getter for anchor tag in URL)

In SPA applications the Router works on top of the last three of the previous list. These three params of the URL are matched by the requested resource by Route matching giving an outcome of the requested resource. React-Router is a package that has been in use in the React Ecosystem for a long time. The version 4.0 is a complete re-write of the popular package and this re-write is a paradigm change towards the Modular / Component model that is currently trending in Web application development. React-Router uses History object in the background for navigation through time, such as using back and forward button. The History object takes a function of event listener as an argument and attach these functions to an array of event listeners. The History object is based upon the Observer pattern, and when location change events occur, it notifies the change of location through navigation to the Components that depend on the History object itself. The React-Router package is divided upon three distinct packages to be used upon the context of environment. These packages are react-router (all-inclusive root package), react-router-dom (package that is browser specific) and react-router-native (package that is used for React-Native mobile development). In the study project we will be using the react-router-dom as the application is meant for browser use only. (Medium Site 2018, Demystifying SPA; Medium Site 2017, React-Router v4 tutorial)

**Styled-Components** is a CSS-in-JS library, and what this means that CSS lives in the Component (JavaScript / React Component) level instead of the DOM level. CSS is not bundled from external CSS file or from any CSS pre-processors such as Sass, Less or the popular PostCSS. The styles themselves attach to the style tag atop of the DOM instead of the traditional inline styles familiar from React development. This means that you get all
of the expressive power that CSS can give you as a developer. With Component inline CSS styles, you are left without CSS pseudo selectors for example. With CSS-in-JS you get several truly wonderful benefits, such as you do not have to write the styles to an external file and the styles can live with your Component, if you choose to do so. No need for bundling CSS with the added complexity and time to compile through bundlers, such as Webpack or Broserify. No Cascading issues, as the CSS lives in Component isolation, as discussed in the Chapter 7.1. Because of the Component isolation, Styled-Components avoid parent to child cascading problems. Vendor prefixing is not needed, because it is automatically implemented already. Styled-Components, as the name suggests is made of just Components, which have to full power of JavaScript. These Components are JS objects, which you can inherit and extend. You make a base Component and, if you want to inherit that, but change something in it, you can just extend it. You can extend everything, such as third-party libraries with styles added to them by wrapping them with Styled-Components. (HackerNoon Site 2017, All you need to know about CSS in JS; Styled-Components GitHub Site 2018, Styled-Components; React Finland DrobBox 2018, Styled-Components)


14 Brief theory behind the concepts of MST

MobX-state-tree is an offspring of parents from different sides of paradigm spectrums. The union of disputing paradigms that are mutable and immutable. The library API is vast in MST (MobX-state-tree), but it is pretty much just an extension of its closest parent library that is MobX and it cannot exist without it in a project. MST loans a lot from Redux, such as the State-tree concept that was mentioned in the chapter 9.3, and what the State-tree concept immutability enables in the form of Time-travel and HMR (Hot-Module-Reloading). There are advantages and disadvantages of using Redux and MobX. The main disadvantage of using Redux is its added boilerplate, but the boilerplate comes with a clear pattern in mind. When you follow this pattern and learn it throughout, you get things done and you are always on top of things happening. MobX on the other hand makes developing easy and fast, as it cuts development time substantiately when compared to Redux. For simple applications MobX is easy to learn and use, but it lacks on patterns and clear usage, as it has no opinions on those from the very beginning. The triumph of freedom can lead to chaos when using MobX with complicated applications, as complex applications usually rely on strong architectures and agreed upon patterns in larger dev teams. I was fascinated about MobX from the very beginning, when I first found out about it. But soon I was aware about its problems and knew that it was just a tool for certain kinds of applications as it did not solve everything in the dilemma of managing the State in React ecosystem. When I found out about MST I was sold right away. What if MobX was opinionated with clear-cut patterns and architecture? What if there was a way to introduce Time-travel, HMR to MobX? Combining the main benefits of MobX and Redux was the reasoning behind MobX-state-tree for Mattia Manzati and Michel Weststrate, and I decided to make my thesis study and the product that I implemented to focus on it.

14.1 State-tree

From mutable tree backed by MobX to structurally shared, serializable, immutable tree. This is how MST can be described, but what are these concepts in detail? Mutability and Reactivity to changes in the data is achieved by the backing of MobX and the concepts of the library has been described in detail in the Chapter 10 and its subchapters. MobX brings the FRP (Functional Reactive Programming) concept to the table and MST leans heavily on the reactivity of MobX in the background to make the changes to be always up to date by observing the data. State is not owned by the mutable tree of MobX, even though it is handled by it. The State is the state owned by MST. The graceful agreement and combination of mutable and immutable tree is pictured in the figure 12. Here the State changes to values are handled by the reactive mutable way with MobX and this produces a snapshot of the current State by the power of computed values described in chapter
10.2.3. This snapshot is the essence of the immutable tree. Every time a CUD (Create, Update, Delete) operation happens in the application, a new snapshot / State-tree gets written. The same thing happens in Redux and this is how MST combines mutability and immutability. (Codeburst Site 2017, The Curious case of mobx-state-tree)

![Diagram: Mutable tree to Immutable tree](Source: Codeburst Site 2017, The Curious case of mobx-state-tree)

**14.1.1 Structural sharing of State-tree**

The theory of structural sharing is what combines such libraries as ReactJS and ImmutableJS. Immutability of React was described for instance in the chapter 6 and just to be clear, React Elements are just objects that are immutable / virtual descriptions of actual DOM elements. The Virtual Dom forms easily traversable tree and the diffing algorithm of React works on it. React reuses / reconciles as many instances as possible and not all instances of React elements are new when the State changes. This helps to preserve the State throughout elements in React. (ReactJS Documentation SITE 2015, React Elements describe a tree; Codeburst Site 2017, The Curious case of mobx-state-tree; MobX-State-Tree GitHub Site 2018, mobx-state-tree)

Structural sharing is something that can happen in a tree shaped object that is usually called as a Trie. Abbreviation of a Trie is a tree and Trie is the ancestor concept that the state tree leans upon. Instead of using just key value mappings, just like in a JS Object or common denominator in computing a hash table, which are represented by Set, Map etc.
in JavaScript, you use a Trie. Trie is a tree shaped object where the root node just stores references to child nodes and is empty otherwise. The child nodes themselves can contain values and references to child nodes and if there aren’t any, then the node will be null. This tree type of a structure makes traversing easy and fast. Traversing is finding values in the Trie by travelling through the branches until the lookup returns a search hit or search miss depending if the value is found in the branches of the Trie / tree. (Medium Site 2017, Trying to understand Tries)

Figure 13: Trie structural sharing
(Source: Medium Site 2016, ImmutableJS - persistent data structures and structural sharing)

Figure 14: Trie with reconciliation
(Source: Medium Site 2016, ImmutableJS - persistent data structures and structural sharing)
The Figure 13 below is a visual representation of a tree, where structural sharing can be seen. The Root node / contains references to the nodes a and c. Nodes a and c themselves contain references to other nodes and they do not contain any values. Only node in this representation, which contains both a value and a reference is the node with the "key" atom. Structural sharing is first seen in the node at and cl, which branch out to different branches. (Medium Site 2016, ImmutableJS - persistent data structures and structural sharing)

Reconciliation is a process where structural sharing shines. This process is visualized in the Figure 14 by the green colors picturing the traversing and reconciliation processes. When new values are added to the tree two things will happen. Firstly, a new tree is built from the root on. Secondly, the existing nodes on the tree that are not affected by the changing values will be reconciled as in reused in the process. (Medium Site 2016, ImmutableJS - persistent data structures and structural sharing; Medium Site 2017, Trying to understand Tries) To use the reconciliation process in full with MST you must use the identifier property type that MST provides. This is suggested and without the property in the node, MST will try to reconcile as many nodes as it can. If the identifier property is not used, then MST tries to use the type system for reconciliation. If the structure of the object in snapshot matches that of the type, then reconciliation will happen. In array structures reconciliation will happen only through the identifier property and, if it is missing in the Model of the store, then those instances in the array are not included in the reconciliation process. (Codeburst Site 2017, The Curious case of mobx-state-tree; MobX-State-Tree GitHub Site 2018, mobx-state-tree)

Mutation is something that is present in JavaScript in default. (Medium Site 2016, ImmutableJS - persistent data structures and structural sharing. Medium Site 2017) Building a new tree every time a change happens in the State by the way of structural sharing and immutability is the way that MST makes the State persistent, as the reconciled nodes keep they're states alive. The State-tree is not mutated in place by values and the original structure is kept intact and a new State is derived from it. All that was not changed in the process, were reconciled and kept to build the new State with the new values to that State. (The Curious case of mobx-state-tree. MobX-State-Tree GitHub Site 2018, mobx-state-tree) React uses this with the diffing algorithm and ImmutableJS as well, which is often paired with Redux development to facilitate immutability. (Medium Site 2016, ImmutableJS - persistent data structures and structural sharing)
14.2 Time travel through Snapshots and patches

MST forms a tree in its own restrictions with a type system to make it reliable, serializable and structurally shared through snapshots. Snapshot is the synonym for generated piece of the State in time that is immutable and thus serializable. Every time the State changes, a new snapshot gets generated. This snapshot generation happens through the backing of MobX computed values. Remember, those computed values that produce no side-effects as they are pure functions and are computed / calculated through the values of observables. These computed values were covered in the Chapter 10.2.3. As snapshots are only the generated, specific points in time, when they were generated, they are stripped to the bone from all other information about what generated them. They are, just like Redux state is, just plain objects and do not contain any other traces of what produced them, such as the MST type system, actions, views etc. Snapshots can be serialized as they are immutable and thus you can send them over the network in an Api request. (Codeburst Site 2017, The Curious case of mobx-state-tree)

Snapshots is only the other side of the coin leaving patches as the other. Patches form the foundation of the Time-travel system in MST. JSON patches is something that MST patches are based upon. JSON patches specification RFC 6902 is a description of changes to a JSON document. This JSON patch is used often with the HTTP method PATCH, when there is a desire to partially update a document and no need to fully update the document. (JSONPatch Site 2018, What is JSON patch). Pathces describe every change in the application values, as all of the values reside at unique paths of their own. These paths are included in the patches, with the description of what changed to what. (Codeburst Site 2017, The Curious case of mobx-state-tree)

14.3 Typesystem forming stores

JavaScript is an multiparadigm language that is interpreted in runtime and not compiled in advance. This has a lot of consequences and one of these is the formation of the type system in JavaScript. JavaScripts type system is a loose dynamic type system, and this works only in runtime. Static type systems such as Facebooks Flow and Microsoft’s TypeScript are supersets of JavaScript that transpile through transcompilation / transpilation step to JavaScript just like JSX, which we did go through in Chapter 4. These supersets bring the OOP pattern of strong typing to JavaScript. (TutorialsPoint Site 2018, TypeScript overview; JCEMER SITE 2017, Types in JavaScript) MST has its own type system that owes a lot to tcomp that is a type system that is supposed to be used during development time. (MobX-State-Tree GitHub Site 2018, mobx-state-tree) The type system of MST ena-
bles type checking in runtime and even design time, when used in combination with TypeScript. Runtime type checking works without the use of TypeScript and it is enabled by all of the MST nodes having a type of their own. (Codeburst Site 2017, The Curious case of mobx-state-tree)

In MST the nodes are objects, which can be types of Model, array and Map and the object primitive properties, which in MST include string, number, integer, Boolean, Date and Custom are considered leaves. Every node in this living tree must exist only once as it is imperative to know that their place is unique. This place where the node lives may be references by non-other than the reference type by another node. You must be able to serialize the leaves in the node. And this means that they cannot be functions for example or other forms that cannot be serialized. (MobX-State-Tree GitHub Site 2018, mobx-state-tree)

Type system forming the stores, which are multiple in MST can be seen in action in the Appendixes 15 – 19. The store / stores are formed from the root type of Model in MST. Every Model is a node that forms its own shape with the primitive leaves or references to other Model types. Each of these Models own their own their state and data. The next chapter will go through the application developed for this study and the details of the type system and managing state with MST will be gone through. (MobX-State-Tree GitHub Site 2018, mobx-state-tree)
15 Application made with MST

The Appendix 2 shows the Client root structure where I have decided to separate the application to models and components folders. Main domain application logic is inhabited by MST stores, which are several and can be visualized in the Appendix 4. The components folder includes the Component elements forming the views and styles folder, which are the styled-components CSS-in-JS styled elements used throughout the application and described in the Chapter 13.2.2.

15.1 Models are stores or stores are models

The MST forming of the state begins from the stores, which are several in MST. Each store definition starts from the MST type model. There are several stores in the application, such as authenticationStore, commandStore, historyStore, queueStore and userStore. We will be focusing on the file commandStore the most in this study, because it is the store where most of the MST type system has been utilized compared to other stores.

15.1.1 AuthenticationStore

Use case for the first store being authenticationStore can be seen in the Chapter 12.2.1 and it lags framework layout view in this study as it did not have one from the very beginning. The materialized view can be seen in Appendix 5.

The authenticationStore is the first store using the MST type system to form a strongly typed node in MST. The Appendix 15 holds the code of the whole store. Here can be seen that store uses two imports from MST and these are the types and flow. First the types.model parameter takes an object that has only primitive types defined in it. The leaves are primitive types, and, in the object, we can see that a shorthand notation is used to define the types in the model. For example, the username property value is left as an empty string, which is a shortcut notation equivalent to types.optional(types.string, ''). The MST has several shorthand notations and two of them are used in the model definition here, one of them I described above and the other one used in this file is the shorthand notation for Boolean seen in for example isAuthenticated property. This shorthand notation also assigns a default value to the property, which in this case is false Boolean value.

In this store we can see the co-location of models, actions and views. Co-location means that through the chainable nature of MST, the actions and model can co-exist / live in the same file unlike Redux where these are defined in separate files.
Action is where the model state is updated in a synchronous fashion. In the authenticationStore we can see the actions function that binds to self, which is always bound and does not suffer from the JavaScript this keyword challenges on bounding to the scope correctly. So, self is this but always bound to the scope and instance where it is proclaimed in. After self we return an object literal with ES6 arrow function right away. Inside this object literal return statement, we have the functions / actions where we modify the state of the node instance / store. The synchronous nature of MST comes from the backing of MobX which was explained in the subchapter of MobX (Chapter 10.2.2) and we can see it in work with the action setIsAuthenticated. The action function setIsAuthenticated takes in the parameter admin in its signature. Then the use of self is first seen in firing the action toggleAuthenticated on self. The action function toggleAuthenticated just toggles the property instance isAuthenticated Boolean value and it is fired in sync and set before going to the second step, which is setting the admin instance property to the parameter admin. After these actions that were fired we can see the sessionStorage write operation that reads from the updated properties of the instance.

The action function submitLogin is where we use flow function we imported from MST. The flow function is a helper method function that eases the handling of asynchronous functions (networking, db calls etc.) in MST. The flow method wraps the ES6 generator function (Mozilla Developer Site 2018, Generator function) in a way that it makes the yield equivalent to async functions await (Async Function. Mozilla Developer Site 2018). You make more with using generators with promises instead of using the normal async function, which always return a promise. The reason to use generators instead of async is that generators can be wrapped to the action functions more easily than the async functions. The yield statement works as equivalent to await still with the flow method, as it was said before. (MobX-State-Tree GitHub Site 2018, mobx-state-tree) The submitLogin function makes an XHR / AJAX (XMLHttpRequest) to the server with axios promise based HTTP library for browsers and NodeJS. (Axios GitHub Page 2018, Axios; Mozilla Developer Site 2018, XMLHttpRequest) This request to the server is yielded and await kind of a style when the response from the server is returned, you can see that if statement is used that depends on the status and success properties got from the response. Depending on the statuses we can set properties or handle errors. When the response from the server indicates a success, we can see that there is a node instance created. We will get back to this later on and clarify why this is possible here.
15.1.2 CommandStore

The commandStore is where most of magic of MST used in this application can be indicated. This study application just scratches all of the possibilities of the library as the requirements of the application prevented me of using them all. The commandStore however shows the full power of both the MST type composition and the use of models as child types.

Appendix 17 shows the full codebase for commandStore. The main models in the commandStore are CommandItem and FastCommandItem, which inherit the main model named Item. To make the model inherit from another object you assign that to a variable and right after that you use the method that is appropriately named as named. By this you get a deeply cloned version of the original object, which return the new type through the named method. The same applies throughout the object, which is not a mutated version of the original object but a deeply cloned and separated object without pointers to the original instance. Props method is used to overwrite the original property constructor and you can see this in the CommandItem and FastCommandItem, which inherit certain common properties in the props method and add their own on top of those. (MobX-State-Tree GitHub Site 2018, mobx-state-tree)

The props method properties include the first place where we can see an example of the type model existing as a property of another model. The parent to child relationship can be seen in the property successful_commands in the CommandItem model. Here we have an optional array of the type of Place. This kind of type composition makes deeply nested object composition with type checking easy and very powerful. You can traverse the deeply nested type composition example with the Place model and see that the place contains child type of Info, which itself has a child type of Row. These models themselves are complex objects containing several properties themselves and this type of type composition with deeply nested objects would be pretty difficult or time consuming to do with for example Redux, where the state object has to be normalized.

There are two action functions that I want to focus on in this chapter, the first one being addLocations action function and the getStatusForCommands action function.

The addLocations function is a part of CommandEditForm. The code for the form can be found in the Appendix 26. In CommandItem action function addLocations we see two parameters, and these are parameters are in fact they're own initialized store instances. The
addLocations action function splits value from location property to values and removes duplicates from it and after it adds them to locations. After that it calls getAddressesForLocations with the stores being as parameters. This type of one-way data-flow by cascading the parameters down by calling actions shines with the synchronicity of MobX. You get action after action, one-by-one and with the power of the backing MobX you know that you are getting a value that is always current.

The second action function we focus on before moving on is the commandSubmit action function. The action is fired when the form is eventually submitted. The action function has several store instances as parameters, one of this being a new one named queueStore. The submitting of the form and the queueStore in action can be seen in the Appendix 8, where there is a one command submitted and pending in the queue seen in the right side of the view. The commandSubmit action function calls an Api with the instance property values. The Use Case for the action function is explained in the Chapter 12.2.3 in the section 4 of the basic flow. Depending on the server response the action function can log the user out, if not authorized, be a recursive function and poll the server in defined intervals of 5 seconds or assign values when the server responds with a body. The MST tree in queueStore forms a new tree in the form of added commands. These commands themselves persist their own protected state and every added command pending keeps polling the server in the same way described above and in the Use Case in Chapter 12.2.3.

15.1.3 queueStore

The queueStore can be seen as the main and root store of the application. The functionality of MST was behind choosing it for this lonely purpose. More on this with the conclusion of the thesis.

The code for queueStore can be seen in the Appendix 19. Here the QueueStore model has property instances / nodes as arrays. For example, the commandList and fastCommandList are typed arrays meaning that these arrays can only inhabit items that are of the type that they are assigned to. The array CommandList can only inhabit the items that are of the type CommandItem and the fastCommandList can only inhabit the type of item that is FastCommandItem.

There are three action functions to this store and these are addCommand, to add a CommandItem and addFastCommand to add FastCommandItem and the last one to destroy the lists when they are not needed anymore.

The computed values in MST are represented by views function on self. The reactivity of MST with MobX comes in handy with this views function. The combinedQueueList must
be a pure function as computed values must be pure. In the combinedQueueList we return a new array with the help of es6 spread operator, which is marked as three dots before the array. This is a shorthand immutable version of es5 concat method and is used throughout the application. Functional programming style with the help of array method sort is chained to array. This sort takes the individual items in the returned array and a function sortDates is applied to them. The sortDates function is a util function in the project imported outside of the scope of this thesis study project, which is the only the Client side. So, I will not add this util file time.js to the project scope in full, but below in the Figure 15 can be seen the sortDates function. The view that holds the queueContainer has the views function, which is sorted every time a command is added to the queue. The newest item by date is always set on top when added to the queue. If the queue would only be sorted by the newest item being on top it would not respect the real order as the arrays were concatenated by the order of the arrays. Doing just the reverse on the array would result in the order of first fastCommandItems and then commandItems. Reordering the items by date we get the real order of the items send. The computed values backed views function is really efficient and consistent in action.

```javascript
sortDates = (first, second, order = 'DESC') => {
    let firstDate = new Date(second.creation_time).getTime();
    let secondDate = new Date(first.creation_time).getTime();
    return order === 'DESC' ? firstDate > secondDate ? 1 : -1 : order === 'ASC' ? firstDate > secondDate ? -1 : 1 : 0;
};
```

Figure 15: sortDates function

15.1.4 HistoryStore

HistoryStore uses every aspect of all of the topics that were discussed in previous model subchapters. Composability, inheritability, models as types, generators wrapped by the flow method and it takes more out of the views function / computed values than any other store in this application.

The Use case for the historyStore is found in the Chapter 12.2.4. The Views for the historyStore include the Appendixes 11 – 15. The code for the store itself is found in the Appendix 17. The application state and the routing throughout several states and screens may seem complicated as the store holds the form for filtering and the history result items as well. Persisting the state properties with the Composable nature of MST and the co-location of the action and views functions was intuitive and made more sense than making this functionality with Redux or MobX for example.

The store holds the form properties for posting a query to the server by the user selected filters. All of the filters have default values on the server, so the only real default values of
interest that the Client itself determines is the starting and ending properties. These values are set by the setStartTime and the equivalent setEndTime action functions. Once again, the reactivity of MST comes handy in setting the state. By using the form in setting the date you soon realize that you can use the date picker functionality of picking a specific date or set the state by yourself through the enabled text-field. This kind of functionality that setting date by hand through the text-field would be very hard to do with Redux for example. The setDate action function has a try-catch block that returns an empty value in the case of an error. This error is caught in every cycle off input when there occurs an error when inputting the date. This kind of functionality catching errors and reacting to them live with just few lines of code justifies MobX and MST over any other library. The validation of the code uses the same approach as the date is only set, if the date is a valid JavaScript date and it passes other validations as well. The value once again is validated by every inputted character.

The real sugar that oozes through views functions can be seen in the getTable function. There the Excel kind of formula returns a pure function that slices an 50 items long list of items from the historyList by the calculating the offset and limit for the slice array method. This React’s to every user action when the paginated inputs shown in the Appendix 13 are clicked. This kind of reactive pagination through the computed values is very efficient and the effectiveness is explained in the Chapter 10.2.3.

15.2 Views

There are several views in the application following the component architecture in utilized in React ecosystem discussed earlier in this thesis. There are several components higher up on the hierarchy connecting to the stores and are therefore stateful. And there are several components that are stateless and the components that are only containers for the style definitions with Styled-Componentss discussed in the Chapter 13.2.2. I will be discussing the concepts in the following subchapters.

15.2.1 Index.js as the entry of the application

The entry of the application is in the index.js file. The code can be found in the Appendix 21. The creation of stores and providing them to the application happens in this file. The application uses mobx-react bindings to bind the stores to the application. This is done through the provider pattern that is imported through the mobx-react package. (MobX-React GitHub Page 2018, mobx-react) Provider pattern is used in Redux as well to connect the React components to the store. Provider pattern uses the context to pass the data to
the child components without explicitly passing them down by props manually in the unidirectional way. (ReactJS Documentation 2018, Context pattern)

The actual creation of the stores happens through the create method on the store instance. The instance is set on a variable and an example of this is shown in the Appendix 21, when commandItem and FastCommandItem stores are created. These items are single instance items, which do not themselves have multiple instances of other stores. By this I mean that these instances are used in other higher up stores in a parent to child relationships. It is better to call them component instances that have their own and hold their local state and this state can be modified by the instance itself or the parent can access that state as well to modify it. The commandItem instance holds the state for the form visualized in the Appendix 7. And the equivalent fastCommandItem holds the state for the form visualized in the Appendix 28.

The other stores are created a little bit differently, which comes apparent immediately when looking at the code. The store create method takes in a util helper function that is imported. The code for the util method can be found in the Appendix 29. The method initializes the store from sessionStorage, if the store is found from it. The utilization of this happens when a user for example pushes the refresh button or F5 on the browser. The other reason for initializing it this way is helping with the development time. Saving the code on the VS text editor with HMR during the development time automatically makes a webpack build configured in webpack.config. The initStore method takes three parameters. The first parameter is used as default values to the store and returned when the sessionStorage does not hold the values for the store. The second parameter is the store itself, which is used to compare the sessionStorage object to the store model type by calling object.is function. The former is a deep type checking on objects. If the sessionStorage object is found in the sessionStorage and it checks out to be the same type as the store, then the store will be initialized from the sessionStorage. With this kind of modification and save cycle, you would lose you’re state in between the cycle. The initStore function that I made helps with this to get the persisted state from the sessionStorage between for instance the development time.

So, the create method takes the initStore method on as a parameter to initialize the store from the sessionStorage between page refreshes. This is a read operation from the sessionStorage, but where is the write operation on it? This is where the snapshot functionality of MST explained in the Chapter 14.3 comes to play for the first time. OnSnapshot function imported from MST used in the index.js file, takes queueStore and every time a
state change happens on the queueStore, a snapshot is generated. This serializable current in time state gets written to the sessionStorage as a result of the onSnapshot function.

The app variable has the Provider element wrapping all of the child elements. Here the store object holding the individual store instances are passed to the application by the Context as described earlier in this Chapter. The application itself holds two possible routes in the beginning by the React Router. These two elements are the Login Component and the actual application, which is the App Component. The actual App is reached after the login is authorized in the Login component and its store, the authenticationStore, which was described in the Chapter 14.3.1.

### 15.2.2 Login Component

The Login Component code can be found in the Appendix 22 and the end result View for the Component is visualized in the Appendix 5 as the first screen for the application. The Login Component is the second main Component in the application with the actual main application Component suitably named as the App Component. The App Component is a wrapper Component holding the main styles of the application and the MainContainer Component, which holds the routing for the application after login. The Login and Main-Container hold the authorization logic routing together. The process goes in the following way. After the Application is mounted, the Switch of React Router in Index.js file looks for a Component that is the root Component. The Root Component is the App Component holding the MainContainer Component and the routing directs to it. The MainContainer Component looks checks the isAuthenticated variable that it gets from the authenticationStore (Chapter 14.3.1). If the isAuthenticated Boolean variable turns out to be false, the user will be redirected to the Login Component. The Login Components logic is similar in a way that it checks the same value isAuthenticated from the authenticationStore. If the isAuthenticated Boolean variable turns out to be false, then the user will be prompt by the Login Component view. If the authorization passes, then the User will be redirected to either the CommandEditForm Component view or the FastEditForm Component view depending on the Boolean value admin in the authenticationStore.

### 15.2.3 MainContainer Component to hold the views of the application

The MainContainer Component resides in the App Component. There are several reasons for this and we have gone through them in the previous Chapter. App Component holds the styles, MobX specific dev tools and the application. Making the App Component as the
The MainContainer Component code can be found in the Appendix 24. The Component holds the entire application views in a React Router Switch, with the exception of the MenuBar Component and the conditionally shown QueueContainer Component. There are all of the main Component imports and all of the store imports found in the file.

There are two imports from mobx-react that are of import. The inject and observer imports are used to wrap a React stateless function that returns a view. There is no need to use a classical react class with MST or MobX as the state is completely handled by these libraries. There is no render function in sight in this file as there is no need to use the optimizations, such as shouldComponentUpdate and also there are no live-cycle methods of React used. The React Components, which mark out the difference of stateless or stateful Components were discussed in the Chapter 7.1.

The inject function is used together with the Provider pattern. The Provider pattern was discussed in the previous Chapter 14.4.1. The Provider pattern springless the content and the state of the store throughout the application. To connect to the store without explicitly passing the props down you must use the inject function method. In the MainContainer Component inject function we can see that the inject takes a rootStore parameter and returns the cherrypicked stores that are used in the Component.

The observer function is another import of mobx-react and with the inject function they form the backbone of MobX bindings to React. The observer was discussed in the Chapter 10.2.1 in more detail. I have chosen not to use the usual way of using MobX decorators to make the Components as observers of the observables. This decorator function does not bring anything of desire, because the Components can be just wrapped inside observer functions. The decorators are only syntactic sugar and the decorators could be left out of future releases of EcmaScript (JavaScript) specifications as they are only on the propositional phase of the language. The observer pattern is what makes the MobX as FRP and, as reactive.

The only real difficult to understand routing method comes with the dynamic routing with the item lists by id’s. The React-Router uses a render props pattern to render the return value of a function as children. (ReactTraining Site 2018, Use a render prop) QueueContainerList in the MainContainer Component is a great example of this and all of the following items that use dynamic routing by id are pretty much the same. The render prop of the

wrapper Component for all of the previously mentioned items made sense at the time. But on the next thing already.
Route element takes a function that matches an item that is the result of an array method find on the list. If the find method result id matches the match method, then the user is routed to that specific item. One result of these dynamic routing results can be found in the Appendix 9, where the view holds a detailed data view for an individual CommandItem (Chapter 14.3.2) from the queueStore (Chapter 14.3.3). If a view is not found or matched a RouteNotFound Component is shown in the behalf of the view.

15.2.4 Editing a Command

As was discussed in previous Chapter (14.4.3), the MainContainer Component withholds all the main component views of the application, that we can route to with React-Router. From this on we will not go through every Component in singled out chapters. Instead we will follow a route through the CommandEdit Component to the CommandEditForm and its children and store actions. This way we will get a specific understanding of what MST and MobX offers in the competition of state management libraries.

The CommandEdit Component code can be found in the Appendix 25. The CommandEdit Component acts as a parent Component to the CommandEditForm and FastCommandEditComponents. The Use case for separating these components to two conditionally accessible components can be found in the Chapters 12.2.2 through 12.2.3. Only one of the components is accessible for the Admin user, and this is the component we will be focusing on, from this on.

CommandEditForm code can be found from the Appendix 26 and the view visualized in the Appendix 7. The Use case for it can be found in the Chapter 12.2.3 and the store functionality through the explicit actions is discussed in the Chapter 14.3.2.

The Component CommandEditForm can be realized as a true stateless Component, which has no local state of its own and it does not connect explicitly to any of the stores. The stores data is passed down as props from the parent Component CommandEdit and the Component itself is wrapped inside a mobx-react observer function. The reason why CommandEditForm default export Component and the Select Component in the same file are wrapped inside an inject function, is the reason behind ensuring MobX reactivity. Even though the child components of a Component may be stateless components themselves, as they are separated from the logic of the functionality and do not alter any of the state in the stores. Even though the previously mentioned, the child Components must still be marked as observers to the observables, as all of the Components that include observable data must be marked so. The marking of the Component as observer kind of makes the component as stateful against the React composable pattern, but with working with
MobX you can forget about this and be sure that this is to make the Component reactive and have always up-to-date data. The optimization that comes with the stateless / dumb Components with React compared to using the observer wrapper in the Components is neglectable in this case. (MobXJS Documentation Site 2018, Observer)

The actions on Component form fields are triggered on inline functions. These actions are explicitly using the store actions in a synchronistic way described in the subchapter of MobX covering the actions with MobX and the previous Chapter of MST 14.3.1.

Removing the local state from React, using MST with MobX backing makes the state tangible as understandable in reacting and serializing it with every state change. MST cuts down on boilerplate of Redux with the same benefits and makes its own opinions how to form the state unlike MobX with the benefits of it. What is not to like? I really feel that with the advancement of MST in the future, it might be the Uniform state management tool. Only time will tell.
16 Conclusion

The objective of this thesis was from the very beginning to take a deep dive into the world and Ecosystem of React theory and particularly its state management with unidirectional data flow. The theory part of this thesis is particularly aimed for developers working at HiQ Finland who are familiar with Client-side web development with ReactJS or equivalent frameworks / libraries. Usually people working with React’s own internal local management system, Redux or MobX are in a state of confusion that comes out of not knowing the basics and theory behind. The theory of this thesis was meant as abstract learning of the theory in writing that justifies the code that is written everyday at the offices of HiQ. What does the code behind do, and what is it based upon? The internet is full of convoluted learning material that does not paint the whole picture. I went through a lot of books for this thesis that were a little bit hacky and did not quite cut it to be added as references for this thesis.

The application developed for a Customer of HiQ is in its testing phase at the current time of writing these lines. The individual instances managing their own states in a way that is referential and persistable was behind the fact of choosing MST (mobx-state-tree) as the State management tool, even when it has not been used in development at HiQ. The choice was the right one in my own opinion, now looking back upon it.

The evaluation of MST for as a force to be reckoned with concerning State management is divided into parts following this conclusion prologue.

16.1 Development with MobX-State-Tree

It all comes down to patterns and architecture that make sense and give value on their own. The applicable feeling of being lost in large applications where multiple developers are working with the very same code base and functionality, can come with working with MobX and Redux. MobX-State-Tree was developed in this purpose in mind, of not being lost. MST with its opinionated structure gives a clear-cut pattern of how to build models, react to changes and persist State.

The learning curve of MST was low for the writer of this thesis and I think this will be the case for senior developers and even beginner / junior developers. Building complex trees / models with references, even circular ones is a breeze compared to other State management tools. Mobx-state-tree brings imperative and functional composable styles together in a way that is intuitive and free of unnecessary boilerplate.
You can develop full applications with MST as the State management tool, but there may be points on the applications functionality where MST could be overkill or MST could add more overhead performance wise. If these points are discovered in the time of performance testing or even better, identified and realized beforehand, you can always fall back on MobX or even React’s own local State and its patterns. MST can be thought of as an extension bridge between Redux and MobX and it is in the end only optional in the project.

Redux devtools can be used with MST out of the box by importing mst-middlewares connectReduxDevTools method. This is used in the index.js file found in the Appendix 21. By connecting to Redux devtools you can use the timetravel undo / redo functionality.

You can also use MobX devtools to optimize rendering of the right elements. For example, the parent should not rerender, if a property is changed only on the child element and vice versa. The reconciliation of element instances and rendering affected is shown through the MobX devtools. The imported DevTools of mobx-react can be found in action in the Appendix 23.

16.2 Testablity of MobX-State-Tree

The concept of snapshots and patches make testability 100 percent coverable in MST. The snapshots make the state serializable out of the box. Every state change is represented by this serialized JSON structure presentation of the state. It is easy to imagine that this presentation of state in snapshots and state changes in patches are comparable in any tests imagined.

The type system with complex types in models and primitive types gives design and runtime assurance in type checking. Enabling TypeScript with MST speeds up development out of the box and gives development time “testability”. The former can be used with VS Code TypeScript plugin and this combined with VS Code JavaScript intellisense is for my mind close to or equivalent to compiled language type checking. This makes MST a no brainer compared to other management tools, such as Redux or MobX.

16.3 MobX and MobX-State-Tree community support

The reason why MobX community is mentioned in the topic of this Chapter is that MobX is what MST is based upon. As an extension library of MobX, MST’s fate is sealed in the ad-
vancement of MobX. MST does not exist as a standalone library and many of its function-
alities could be merged to the parent libraries framework. The fate of MST could be, the 
fate of Homo neanderthalensis that were absorbed by Homo sapiens. The former mean-
ing that MST could be fused with MobX at some point.

The community of MST is a small community at this point. The coverage of MST’s func-
tionality in the API docs is vast and combined with the MobX community and API docs you 
can get an answer even to the most hardest of problems to be solved.

16.4 Added value HiQ and to the Customers

Cutting down development time with React is a constant threshold question in the 
frontend community and in ICT industry. Added time that comes with a lot of boilerplate is 
usually essential in quality assurance with the frameworks such as Redux. Testability, de-
buggability and reliability that comes with Redux has been outmatched this far. This has 
lead to a situation where the competition with large applications has been stale for a few 
years. Due to tangible JavaScript fatigue in the industry, Redux has been the de-facto 
tool concerning State management, even though the obvious fact of added boilerplate and 
therefore added development time.

The other approach of using the competitor MobX has been the appealing alternative of 
State management tools of late. MobX however lacks in patterns and opinions of how and 
this may lead to problems with frontend developer teams exceeding two developers.

Quality assurance and the efficiency that comes from developing with MST, is in my mind 
uncomparable. The paradigm needs not to shift from imperative to functional. The para-
digm needs to fuse. As it is said, MST is a mutable / immutable tree, the combination of 
usually competitive paradigms. A fusion of functional and imperative. A fusion of Quality 
Assurance and efficiency.

16.5 Outcomes of learning

As an FullStack developer I have been in the roles of both Frontend developer and 
Backend developer in projects ranging from small to large. This project was the first where 
I was responsible as a sole resource FullStack developer for an application in its all devel-
opment cycles. I was also the sole responsible person for the architecture, UX design and 
devops. The whole process of holding all the threats in hand as a puppeteer was very 
educational and gratifying. I am grateful for the opportunity that I got from HiQ to do this
project and as well to base my thesis upon one of the crucial building blocks of the application in the form of MST.

Drilling my way through countless sources of information on State management and revising my knowledge on subjects such as JavaScript, React Ecosystem, DOM or HTML in general was invaluable at this point of my career. Learning React, is not learning about just a framework or a library, it is learning about the Ecosystem needed to build applications in a modular way with React and the language of JavaScript and the newer EcmaScript standardizations of the language.

I am sure that MST brings an added value to HiQ in the form of development speed, testability and QA. This thesis and the application developed to one of HiQ’s customers will act as a proof-of-concept to the power of the union between mutability and immutability in the form of MST.
17 Summary

The problem of patterns and paradigms on Server-side development has been solved for years to come. The emergence of modular, component patterns that was enabled on the Client-side by React has enhanced Client-sides usefulness and shifted the management of data towards it. React is just a view library concerned about rendering elements. It is not a full blown, all-inclusive MVC framework and with React, you build a framework of your own.

React's own local State management is sufficient with small applications and all developers should use it and learn it on their own, to see what makes it tick. Patterns used with React's own State management may lead to cumbersome boilerplate and hard to deal with bugs. Larger applications need something more to make the data flow down unidirectional way. This is where Flux pattern and frameworks come in. This pattern is the base upon many of the frameworks that manage the state with React and bring the M (model) and C (controller) to complement the V (view).

The triumphant framework that came from the wars of Flux was Redux. This framework has now stood as the de-facto tool of State management for years. The paradigm of functional programming came with it. It shocked the JavaScript community with the paramount introduction of immutability and brought it to the lunch conversation tables. This is what the cool kids of JS community have been raving about for years.

The one that took a leap from the shadows to the lime light as contender was MobX. The familiar good old OOP concepts felt imperative again in a cool way. The paradigm of TFRP (Transactional Functional Reactive Programming) cut down most of the forest of boilerplate and made it in a way that guaranteed the data to be current.

MST (mobx-state-tree) the new kid on the block, counseled a truce between Redux and MobX and uses a combination of the best aspects of these in mutable / immutable way. MST was chosen as the State management for an application that was developed for a Customer of HiQ. This proof-of-concept application (that is in its state of application in waiting / test) opened up the possibilities of MST.

The State is made serializable through Kodak moment like snapshots, which make it tangible. This tangible / serializable state through snapshots and patches are 100 percent testable through, such testing tools as expect.js. Redux devtools through the browser are
something that can be used with the combination of MobX devtools. With these you can be always in point of what is happening with your application.

The type system of run-time and design time type checking is a powerful tool for loosely type language as JavaScript. TypeScript is optional, and you can leave it out, as I have in the project development time. Runtime, development time type checking is still a powerful addition to behold.

The opinions of how to build your models are strict but rich in the same time with MST. Complex models with references to other instances, even cyclic ones can be achieved with ease.

The main point of MST is that, you do not need to choose between the paradigms of FP (functional programming) or imperative / reactive data, you can choose both.
References


Medium Site 2016, What is VirtualDOM. URL: https://medium.com/@tony_freed/what-is-virtual-dom-c0ec6d6a925c. Author: Tony Freed. Accessed: 8 March 2017


W3Schools Site 2018, JS Versions. URL: https://www.w3schools.com/js/js_versions.asp. Accessed: 21 October 2018

Appendices

Appendix 1. The Project structure on root level

Appendix 2. The Client root structure
Appendix 3. The Client Component structure

```markdown
client
  components
    styles
      App.js
      CommandEdit.js
      CommandEditForm.js
      CommandItemView.js
      DatePicker.js
      Details.js
      FastEditForm.js
      FilterContainer.js
      HistoryFilterForm.js
      HistoryItemView.js
      HistoryResult.js
      HistoryTable.js
      Info.js
      LocationList.js
      Locations.js
      Login.js
      MainContainer.js
      MenuBar.js
      NotFound.js
      QueueContainer.js
      QueueItemView.js
      SelectedFilterBox.js
    models
    static
    index.html
    index.js
    {} jsconfig.json
```
Appendix 4. The Client Model / Stores structure
Appendix 5. The Login page

Appendix 6. Landing page for Administrator
Appendix 7. Command screen

Appendix 8. One Command request in Queue
Appendix 9. Successful and unsuccessful Commands
Appendix 10. Details of one Command

![Image of data table]

<table>
<thead>
<tr>
<th>Type</th>
<th>Area</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR</td>
<td>501</td>
<td>10%</td>
</tr>
<tr>
<td>T1</td>
<td>581</td>
<td>10%</td>
</tr>
<tr>
<td>T2</td>
<td>823</td>
<td>10%</td>
</tr>
<tr>
<td>Y5</td>
<td>796</td>
<td>10%</td>
</tr>
<tr>
<td>T4</td>
<td>923</td>
<td>10%</td>
</tr>
</tbody>
</table>
Appendix 11. History Filter page
Appendix 12. History search result index screen
Appendix 13. History search result paginated result tables

<table>
<thead>
<tr>
<th>Käyttöpaikka</th>
<th>Osoite</th>
<th>Operaatio</th>
<th>Aloitettu</th>
<th>Lopettettu</th>
<th>Käyttäjä</th>
<th>Tila</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>Obusikkeentie 7 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 15:08</td>
<td>02.08.2018 15:08</td>
<td>voad</td>
<td>X</td>
</tr>
<tr>
<td>103</td>
<td>Obusikkeentie 9 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 15:08</td>
<td>02.08.2018 15:08</td>
<td>voad</td>
<td>✓</td>
</tr>
<tr>
<td>105</td>
<td>Obusikkeentie 4 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 15:08</td>
<td>02.08.2018 15:08</td>
<td>voad</td>
<td>✓</td>
</tr>
<tr>
<td>106</td>
<td>Obusikkeentie 8 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 15:08</td>
<td>02.08.2018 15:08</td>
<td>voad</td>
<td>X</td>
</tr>
<tr>
<td>102</td>
<td>Obusikkeentie 7 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 12:42</td>
<td>02.08.2018 12:42</td>
<td>maaju</td>
<td>✓</td>
</tr>
<tr>
<td>103</td>
<td>Obusikkeentie 9 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 12:42</td>
<td>02.08.2018 12:42</td>
<td>maaju</td>
<td>✓</td>
</tr>
<tr>
<td>104</td>
<td>Suonatie 16 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 12:42</td>
<td>02.08.2018 12:42</td>
<td>maaju</td>
<td>✓</td>
</tr>
<tr>
<td>105</td>
<td>Obusikkeentie 4 06100 PORVOD</td>
<td>Intägykentä</td>
<td>02.08.2018 12:42</td>
<td>02.08.2018 12:42</td>
<td>maaju</td>
<td>✓</td>
</tr>
</tbody>
</table>
Appendix 14. Details of one History Result Item

### HISTORIA

- **Komennon suorittaja**: vend
- **Toiminta**: liitukentti
- **Syy**: Mauto
- **Käytöpaikka**: 102
- **Osoite**: ChildCare: 7 05100 Pori
- **Loayta**: 02.08.2018 15:08
- **Suoritettu**: 02.08.2018 13:08
- **Kokonaisaste**: 0 s

#### Listitiedot

<table>
<thead>
<tr>
<th>Tarkistus</th>
<th>Areena</th>
<th>Ykköseké</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>551</td>
<td>kWh</td>
</tr>
<tr>
<td>T1</td>
<td>591</td>
<td>kWh</td>
</tr>
<tr>
<td>T2</td>
<td>123</td>
<td>kWh</td>
</tr>
<tr>
<td>T3</td>
<td>556</td>
<td>kWh</td>
</tr>
<tr>
<td>T4</td>
<td>515</td>
<td>kWh</td>
</tr>
</tbody>
</table>
Appendix 15. Fast Command page

Appendix 16. AuthenticationStore model code
setPassword(event) {
    self.password = event.target.value;
},
toggleAuthenticated() {
    self.isAuthenticated = !self.isAuthenticated;
},
setIsAuthenticated(admin) {
    self.toggleAuthenticated();
    self.admin = admin;
    window.sessionStorage.setItem('authenticationStore', JSON.stringify({
        isAuthenticated: self.isAuthenticated,
        admin: self.admin
    }));
},
setLoadState(state) {
    self.state = state;
},
resetCredentials() {
    self.username = '';
    self.password = '';
},
setMessage(message) {
    self.reqMessage = message;
},
submitLogin: flow(
    function* loginUser() {
        try {
            const config = {
                method: 'post',
                url: ENDPOINT_AUTH,
                data: {
                    username: self.username,
                    password: self.password
                }
            };
            const {
                status,
                success,
                message,
                user,
                token,
                admin
            } = yield axios(config)
            .then(r => r.data)
            .catch(e => e.response.data)
            if(success && status === 200) {
                self.setLoadState('done');
                self.setIsAuthenticated(admin);
            }
        } catch (e) { throw e; }
    })
}
Appendix 17. CommandStore model code for Command and FastCommand

```javascript
const timeout = 3000;

import { types, flow, clone, addMiddleware } from 'mobx-state-tree';
import uuid from 'uuid/v4';
import axios from 'axios';
import { curFinnishTimeIso } from '../../../../../util/time'
import {
  ENDPOINT_STATUS,
  ENDPOINT_COMMAND,
  ENDPOINT_FAST,
  ENDPOINT_ADDRESS
} from '../../../../../constants';

```
const Row = types.model({
  Phase: types.maybe(types.string),
  Voltage: types.maybe(types.string),
  Current: types.maybe(types.string),
  Time: types.maybe(types.string),
  Jännnite: types.maybe(types.string),
  Virta: types.maybe(types.string),
  Tariffi: types.maybe(types.string),
  Arvo: types.maybe(types.string),
  Yksikkö: types.maybe(types.string)
});

export const Info = types.model({
  command_id: 0,
  command: '',
  address: types.maybe(types.string),
  command_reason: types.maybe(types.string),
  status: '',
  performer: '',
  command_created: '',
  performed_time: types.maybe(types.string),
  planned_perform_time: types.maybe(types.string),
  headers: types.optional(types.array(types.optional(types.string, ''))), []
    rows: types.optional(types.array(Row), [])
});

const Place = types.model({
  usage_place_id: '',
  command_id: 0,
  info: types.maybe(Info),
  info_to_show: types.maybe(Info)
})

.actions(self => {
  showInfo(info) {
    console.log('show info')
    const infoClone = clone(info);
    self.info_to_show = infoClone;
  },
  resetInfoToShow() {
    console.log('close info')
    self.info_to_show = null;
  }
});

const Item = types.model({
  user: types.optional(types.string, ''),
  location: types.optional(types.string, ''),
  creation_time: types.optional(types.string, '')
})
ending_time: types.optional(types.string, "."),
state: types.optional(types.enumeration('State', ['pending', 'done', 'error']), 'pending'),
message: ",",
request_duration: types.optional(types.string, ")
})

.actions(self => {{
    setLocation(event) {
        self.location = event.target.value;
    },
    setLoadState(state) {
        self.state = state;
    },
    setMessage(message) {
        self.message = message;
    },
    setRequestDuration(ending_time, request_duration) {
        self.ending_time = ending_time;
        self.request_duration = request_duration;
    },
    showInfo(message, state) {
        self.setLoadState(state);
        self.setMessage(message);
        setTimeout(() => {
            self.setMessage('');
            self.setLoadState('pending');
        }, timeout);
    },
    showError(message) {
        self.setLoadState('error');
        self.setMessage(message);
        setTimeout(() => {
            self.setMessage('');
            self.setLoadState('pending');
        }, timeout);
    },
    setFinalState(message, state) {
        self.setMessage(message);
        self.setLoadState(state);
        setTimeout(() => {
            self.setMessage('');
            self.setLoadState('pending');
        }, timeout)
    }
}});

const LocationWithAddress = types.model({
    usage_place_id: ",",
    address: "
});

export const CommandItem = Item
    .named('CommandItem')
.props({
  id: types.optional(types.string, uuid()),
  operation: types.optional(types.string, ""),
  reason: types.optional(types.string, ""),
  locations: types.optional(types.array(types.string), []),
  successful_commands: types.optional(types.array(Place), []),
  failed_commands: types.optional(types.array(Place), []),
  locations_state: types.optional(types.enumeration("State", ['pending', 'done', 'error', ""]), ""),
  locations_with_addresses: types.optional(types.array(LocationWithAddress), [])
})

.actions(self => {
  addMiddleware(self, (call, next, abort) => {
    // Only for logging purposes for now
    if (call.name === 'getStatus' && call.context.state === 'done') {
      // console.log('Done', call)
      // abort(call);
      next(call);
    } else if (call.name === 'getStatus') {
      // console.log(call)
      next(call);
    } else {
      next(call);
    }
  });

  return {
    addLocations(userStore, authenticationStore) {
      if (self.location.length >= 2) {
        let locationsWithinLocation = self.location.split(" ").filter(s => s !== "");
        const duplicatesRemoved = new Set([...self.locations, ...locationsWithinLocation]);
        self.locations = Array.from(duplicatesRemoved).slice(0, 50);
        self.location = "";
        self.getAddressesForLocations(userStore, authenticationStore);
      }
    },
    setLocationsWithAddresses(locations) {
      self.locations_with_addresses = [...locations];
    },
    resetLocationsState() {
      self.locations_state = "";
    },
    getAddressesForLocations: flow(function* getAddresses(userStore, authenticationStore) {
      const token = userStore.token;
      self.locations_state = 'pending';
      try {
        const config = {
          method: 'post',
          headers: {
            // Additional headers if needed
          }
        }
      }
    })
  }
})
authorization: `Bearer ${token}`,

url: ENDPOINT_ADDRESS,
data: {
  locations: self.locations
}

const { success, status, message, responseBody } = yield axios(config)
  .then(r => r.data)
  .catch(error => {
    if (error.response) {
      return error.response.data;
    } else if (error.request) {
      return {
        status: 408,
        success: false,
        message: 'Could not get any response from the server'
      };
    } else {
      return {
        status: 500,
        success: false,
        message: 'Unexpected error occurred'
      };
    }
  });

if (success && status === 200) {
  self.locations_state = 'done';
  self.setLocationsWithAddresses(responseBody);
  setTimeout(() => {
    self.resetLocationsState();
  }, 3000)
} else if (status === 401 || status === 403) {
  userStore.forgetUser();
  authenticationStore.logout();
}
else {
  self.locations_state = 'error';
  setTimeout(() => {
    self.resetLocationsState()
  }, 3000)
```javascript
} , 3000) }
)
} catch (error) {

    self.locations_state = 'error';

    setTimeout(() => {
        self.resetLocationsState();
    }, 3000)
}

setOperation(event) {
    self.operation = event.target.value;
},

setReason(event) {
    self.reason = event.target.value;
},

removeLocation(location, index) {
    self.locations.splice(location, 1);
},

removeLocationWithAddress(location, index) {
    self.locations_with_addresses.splice(location, 1);
    self.removeLocation(location, index);
},

AddLocationOnEnter(e, userStore, authenticationStore) {
    if (e.key === 'Enter') {
        e.preventDefault();
        e.stopPropagation();

        self.addLocations(userStore, authenticationStore);
    };
},

resetCommand() {

    self.id = uuid();
    self.operation = '';
    self.reason = '';
    self.locations = [];
    self.locations_with_addresses = [];
    self.creation_time = '';

},

setCommandLists(successfull, failed) {

    self.successful_commands = [...successfull];
    self.failed_commands = [...failed];

},
```
getStatusForCommands: flow(
    function* getStatus(token, commandItem) {

        try {
            const config = {
                method: 'post',
                headers: {
                    authorization: `Bearer ${token}`,
                },
                url: ENDPOINT_STATUS,
                data: {
                    usage_places: commandItem.successful_commands,
                    creation_time: commandItem.creation_time
                }
            };

            const {  
                success,
                status,
                message,
                successful_commands,
                failed_commands,
                ending_time,
                request_duration
            } = yield axios(config)
                .then(r => r.data)
                .catch(error => {
                    if (error.response) {
                        return error.response.data;
                    } else if (error.request) {
                        return {
                            status: 408,
                            success: false,
                            message: 'could not get any response from the server'
                        };
                    } else {
                        return {
                            status: 500,
                            success: false,
                            message: 'unexpecter error occurred'
                        };
                    }
                });

                if (success && status === 200) {
                    setTimeout(() => {
                        self.getStatusForCommands(token, commandItem)
                    }, 10000)
                } else if (success && status === 201) {
            }
        }
    })
);
commandItem.setCommandLists(successful_commands, failed_commands);
commandItem.setRequestDuration(ending_time, request_duration);
self.showInfo(message, 'done');
commandItem.setFinalState(message, 'done');

} else {

commandItem.setCommandLists(successful_commands, failed_commands);
commandItem.setFinalState(message, 'error');

};
} catch (error) {

commandItem.setCommandLists([], [
    ...commandItem.successful_commands,
    ...commandItem.failed_commands
]);

commandItem.setFinalState('Unexpected error occurred', 'error');

};
}

commandSubmit: flow(
  function* submit(e, queueStore, userStore, authenticationStore) {
    try {

      e.preventDefault();
      e.stopPropagation();

      self.creation_time = curFinnishTimeIso();
      const token = userStore.token;

      const config = {
        method: 'post',
        url: ENDPOINT_COMMAND,
        headers: {
          authorization: `Bearer ${token}`,
        },
        data: {
          command: self.operation,
          perform_time: self.creation_time,
          command_reason: self.reason,
          usage_place_ids: self.locations
        }
      }

    } catch (error) {

      commandItem.setCommandLists([], [commandItem.successful_commands, commandItem.failed_commands]);

      commandItem.setFinalState('Unexpected error occurred', 'error');

    };
  }
);
const {
    status,
    success,
    message,
    successful_commands,
    failed_commands
} = yield axios(config)
    .then(r => r.data)
    .catch(e => e.response.data)

if (success && status === 200) {
    self.showInfo(message, 'pending');
    self.setCommandLists(successful_commands, failed_commands);
    const commandItem = clone(self);
    self.resetCommand();
    queueStore.addCommand(commandItem);

    setTimeout(() => {
        self.getStatusForCommands(token, commandItem);
    }, 5000)
} else if (status === 401 || status === 403) {
    userStore.forgetUser();
    authenticationStore.logout();
}

} else {
    self.showError(message);
};

self.resetCommand();

} catch (error) {
    self.showError('Unexpected error occurred' + error);
};

export const FastCommandItem = Item
    .named('FastCommandItem')
    .props(
        id: types.optional(types.string, uuid()),
location_to_submit: types.optional(types.string, ""),
operation: 'current_voltage',
info: types.maybe(Info)
})
.actions(self => {
  setLocation(event) {
    self.location = event.target.value.replace(/\s/g, "");
  },
  AddLocationOnEnter(e) {
    if (e.key === 'Enter') {
      e.preventDefault();
      e.stopPropagation();
      self.addLocationToSubmit(e);
    }
  },
  addLocationToSubmit(e) {
    if (self.location.length >= 2) {
      self.location_to_submit = self.location;
      self.location = "";
    }
  },
  removeLocationToSubmit() {
    self.location_to_submit = ""
  },
  resetFastCommand() {
    self.id = uuid();
    self.location = "";
    self.location_to_submit = "";
    self.creation_time = "";
  },
  setInfo(info) {
    self.info = info;
  },
  fastCommandSubmit: flow(
    function* submit(e, queueStore, userStore, authenticationStore) {
      self.creation_time = curFinnishTimeIso();
      const fastCommandItem = clone(self);
      queueStore.addFastCommand(fastCommandItem);
      self.resetFastCommand();
      try {
        e.preventDefault();
        e.stopPropagation();
      }
    }
  )
const token = userStore.token;

const config = {
    method: 'post',
    url: ENDPOINT_FAST,
    headers: {
        authorization: `Bearer ${token}`,
    },
    data: {
        command: fastCommandItem.operation,
        usage_place_id: fastCommandItem.location_to_submit,
        creation_time: fastCommandItem.creation_time
    }
};

const {
    status,
    success,
    message,
    info,
    ending_time,
    request_duration
} = yield axios(config)
    .then(r => r.data)
    .catch(e => e.response.data);

if (success && status === 200) {
    fastCommandItem.setInfo(info);
    fastCommandItem.setRequestDuration(ending_time, request_duration);
    self.showInfo(message, 'done');
    fastCommandItem.setFinalState(message, 'done');
}

} else if (status === 401 || status === 403) {
    userStore.forgetUser();
    authenticationStore.logout();
}

} else {
    fastCommandItem.setFinalState(message, 'error');
};
}

} catch (error) {
    fastCommandItem.setFinalState('Unexpected error occurred', 'error');
}

};
});
import { types, flow } from 'mobx-state-tree';
import axios from 'axios';
import { ENDPOINT_HISTORY } from '../..//constants';
import { curFinnishTimeIso } from '../..//util/time';
import moment from 'moment';
import { ENDPOINT_HISTORY } from '../..//constants';
import { fiveYearsAgo, oneHourFromNow } from '../..//util/time';
import { Info } from './commandStore';
import { statusList, operationList } from '../..//constants';

export const HistoryItem = Info.
  .named('HistoryItem')
  .props({
    usage_place_id: types.optional(types.string, ''),
    address: types.maybe(types.string),
    request_duration: types.optional(types.string, '')
  });

export const HistoryStore = types
  .model('HistoryStore', {
    historyList: types.array(HistoryItem),
    state: types.optional(types.enumeration('State', ['pending', 'done', 'error', '']), ''),
    message: types.optional(types.string, ''),
    locationsFilter: types.optional(types.array(types.string), []),
    operationsFilter: types.optional(types.array(types.string), []),
    statusFilter: types.optional(types.array(types.enumeration('Status', ['all', 'ok', 'error', 'pending', ''])), []),
    usersFilter: types.optional(types.array(types.string), []),
    location: types.optional(types.string, ''),
    operation: types.optional(types.string, ''),
    user: types.optional(types.string, ''),
    status: types.optional(types.string, ''),
    starting: types.optional(types.string, ''),
    ending: types.optional(types.string, ''),
    creationTime: types.optional(types.string, ''),
    activePage: 1,
    numberOfItemsToShow: 50,
  })
  .actions(self => {
    setOperation(e) {
      self.operation = e.target.value;
    },
    setLocation(event) {
      self.location = event.target.value;
    },
    setUser(event) {
      self.user = event.target.value.replace(/\s/q, '');
    }
  });
setStatus(e) {
    self.status = e.target.value;
},

setDate(date) {
    try {
const dateToValidate = moment(date);
const isValid = dateToValidate.isValid();
// The date needs to be valid moment date and at the most five years ago and hour
from now
if (isValid && date > fiveYearsAgo && date < oneHourFromNow) {
    const selectedDate = date.toISOString(true).split('+')[0].concat('Z');
    return selectedDate;
} else {
    return '';  
}
} catch (error) {
    return '';  
}
},

setStartTime(date) {
    self.starting = self.setDate(date);
    self.ending = !self.ending ? curFinnishTimelso() : self.ending;
},

setEndTime(date) {
    self.starting = !self.starting ? self.setDate(moment().startOf('day')) : self.starting;
    self.ending = self.setDate(date);
},

addLocations() {
    if (self.location.length >= 2) {
let locationsWithinLocation = self.location.split(',').filter(s => s !== '');
const filterList = new Set([...self.locationsFilter, ...locationsWithinLocation]);
self.locationsFilter = Array.from(filterList);
self.location = '';
    }
},

removeLocation(index) {
    self.locationsFilter.splice(index, 1);
},

addOperations() {
if (self.operation === 'all') {
    self.operationsFilter = operationList.map(s => s.en);
    self.operation = "";
}

} else {
    const filterList = new Set([self.operation, ...self.operationsFilter]);
    self.operationsFilter = Array.from(filterList);
    self.operation = "";
}
},
removeOperation(index) {
    self.operationsFilter.splice(index, 1);
},
addUsers() {
    if (self.user.length >= 3) {
        const filterList = new Set([self.user, ...self.usersFilter]);
        self.usersFilter = Array.from(filterList);
        self.user = "";
    }
},
removeUser(index) {
    self.usersFilter.splice(index, 1);
},
addStatus() {
    if (self.status === 'all') {
        self.statusFilter = statusList.map(s => s.en);
        self.status = "";
    }
} else {
    const filterList = new Set([self.status, ...self.statusFilter]);
    self.statusFilter = Array.from(filterList);
    self.status = "";
}
},
removeStatus(index) {
    self.statusFilter.splice(index, 1);
addOnEnter(e) {
    if (e.key === 'Enter') {
        e.preventDefault();
        e.stopPropagation();
        self.addLocations();
        self.addUsers();
    }
}

setHistoryList(list) {
    self.historyList = [...list];
}

submitFilterRequest: flow(
    function* submit(e, userStore, authenticationStore) {
        e.preventDefault();
        e.stopPropagation();

        try {
            const token = userStore.token;
            self.creation_time = curFinnishTimeIso();
            self.starting = self.starting || moment().startOf('day').toISOString(true).split('+')[0].concat('Z');
            self.ending = self.ending || curFinnishTimeIso();
            self.state = 'pending';

            const config = {
                method: 'post',
                url: ENDPOINT_HISTORY,
                headers: {
                    authorization: `Bearer ${token}`,
                },
                data: {
                    commands: self.operationsFilter,
                    performers: self.usersFilter,
                    usage_place_ids: self.locationsFilter,
                    status: self.statusFilter,
                    interval_start: self.starting,
                    interval_end: self.ending,
                    creation_time: self.creation_time
                }
            };

            const { status, success, message,
history_list,
request_duration
) = yield axios(config)
  .then(r => r.data)
  .catch(e => e.response.data)

if (success) {
  self.setHistoryList(history_list);
  self.message = message;
  self.state = 'done';
}
} else if (status === 401 || status === 403) {
  userStore.forgetUser();
  authenticationStore.logout();
}
} else {
  self.message = message;
  self.state = 'error';
}
} catch (error) {
  self.state = 'error';
  self.message = 'Odottamaton virhe tapahtui. Ota yhteys palveluntarjoajaan!'
}

handlePageChange(pageNumber) {
  self.activePage = pageNumber;
},
resetFilters(e) {
  e.preventDefault();
  e.stopPropagation();
  self.historyList = [];
  self.locationsFilter = [];
  self.operationsFilter = [];
  self.statusFilter = [];
  self.usersFilter = [];
  self.location = "";
  self.operation = "";
  self.user = "";
  self.status = "";
  self.starting = "";
  self.ending = "";
```javascript
self.creationTime = '';
self.activePage = 1;
self.state = '';
}
})
.views(self => {

get unfinishedCommandCount() {
    return self.commands.filter(command => !command.finished).length;
},
get getTable() {

    const offSet = (self.activePage - 1) * self.numberOfItemsToDisplay;
    const limit = offSet + self.numberOfItemsToDisplay;
    return [...self.historyList].slice(offSet, limit);
},
get selectedCount() {
    const offSet = (self.activePage - 1) * self.numberOfItemsToDisplay;
    return self.historyList.length > self.numberOfItemsToDisplay ?
        `$({
            (offSet + self.numberOfItemsToDisplay) > self.historyList.length ?
            `${offSet} - ${self.historyList.length}`
        :
        `${offSet} - (${offSet + self.numberOfItemsToDisplay})` / ${self.historyList.length}`
    :
    '0 - ${self.historyList.length}'
}
})
```

Appendix 19. QueueStore Model Code

```javascript
import { types } from 'mobx-state-tree';
import { CommandItem, FastCommandItem } from './commandStore';
import { sortDates } from '.../util/time'

export const QueueStore = types
    .model('QueueStore', {
        queueList: types.array(CommandItem),
        commandList: types.optional((types.array(CommandItem)), []),
        fastCommandList: types.optional(types.array(FastCommandItem), [])
    })
.views(self => {
    get combinedQueueList() {
        return [
            ...self.commandList,
            ...self.queueList,
            ...self.fastCommandList
        ].sort((a, b) => sortDates(a, b));
    },

```
Appendix 20. UserStore Model Code

```javascript
import { types } from 'mobx-state-tree';

export const UserStore = types.model(
  {
    username: types.optional(types.string, ''),
    token: types.optional(types.string, '')
  }
).actions(self => {
  setUser: (username, token) => {
    window.sessionStorage.setItem('userStore', JSON.stringify({ username, token }));
    self.username = username;
    self.token = token;
  },
  forgetUser: () => {
    self.username = '';
    self.token = '';
  }
});
```

Appendix 21. Index.js as Client application root

```javascript
//DEPENDENCIES
import React from 'react';
import ReactDOM from 'react-dom';
import * as mobx from 'mobx';
mobx.useStrict(true);
import { Provider } from 'mobx-react';
import { onSnapshot } from 'mobx-state-tree';
import { Router, Route, Switch } from 'react-router-dom';
import { connectReduxDevtools } from "mst-middlewares";
import createBrowserHistory from 'history/createBrowserHistory';
const history = createBrowserHistory();
//TOOLS
```
import { initStore } from './util/persist';

//COMPONENTS
import App from './components/App';
import Login from './components/Login';
import AppShell from './components/styles/AppShell';
import { RouteNotFound, CaptureRouteNotFound } from './components/Notfound';

//UTIL
import normalize from './util/normalize';

//ICONS
import fontawesome from '@fortawesome/fontawesome';
import {
  faCog,
  faArrowLeft,
  faPlusCircle,
  faTimesCircle,
  faUserSecret,
  faCaretDown,
  faCaretUp,
  faSearch,
  faInfoCircle,
  faExclamationTriangle
} from '@fortawesome/fontawesome-free-solid';

fontawesome.library.add(
  faCog,
  faArrowLeft,
  faPlusCircle,
  faTimesCircle,
  faUserSecret,
  faCaretDown,
  faCaretUp,
  faSearch,
  faInfoCircle,
  faExclamationTriangle
);

//STORES
import {
  CommandItem,
  FastCommandItem,
  HistoryStore,
  QueueStore,
  AuthenticationStore,
  UserStore
} from './models/index';

//Function to initialize a store from sessionStorage.
//Params: initializable object, the store to initialize to check
//if the store datamodel is changed so sessionStorage data is wiped out
//storage type for name in the sessionStorage.
/No need to save CommandItem & FastCommandItem to the sessionStorage as they are form data to be submitted...

```javascript
const commandItem = CommandItem.create();
const fastCommandItem = FastCommandItem.create();

let queueStore = QueueStore.create(
  initStore(
    {
      queueList: [],
      commandList: [],
      fastCommandList: []
    },
    QueueStore,
    'queueStore'
  )
);

const historyStore = HistoryStore.create(
  initStore(
    {
      historyList: []
    },
    HistoryStore,
    'historyStore'
  )
);

const userStore = UserStore.create(
  initStore({},
    UserStore,
    'userStore'
  )
);

const authenticationStore = AuthenticationStore.create(
  initStore({},
    AuthenticationStore,
    'authenticationStore'
  )
);

onSnapshot(queueStore, (snapshot) => {
  window.sessionStorage.setItem('queueStore', JSON.stringify(snapshot));
});

const store = {
  queueStore: queueStore,
  historyStore: historyStore,
  commandItem: commandItem,
  fastCommandItem: fastCommandItem,
  authenticationStore: authenticationStore,
```
userStore: userStore

// Using reduxDevtools in dev for mobx-state-tree
if(process.env.NODE_ENV !== 'production') {
  window.__store = historyStore;
  connectReduxDevtools(require('remotedev'), historyStore);
}

const app = (
  <Provider {...store}>
    <Router history={history}>
      <CaptureRouteNotFound>
        <AppShell>
          <Switch>
            <Route path='/login' component={Login} />
            <Route path='/' component={App} />
            <RouteNotfound />
          </Switch>
        </AppShell>
      </CaptureRouteNotFound>
    </Router>
  </Provider>
);

const render = () => {
  normalize();
  ReactDOM.render(
    app,
    document.getElementById('app')
  );
};

render();

if (module.hot) {
  module.hot.accept([app], () => {
    // new components
    render()
  })

  module.hot.accept(['./models/queueStore'], () => {
    // new model definitions
    const snapshot = getSnapshot(QueueStore);
    queueStore = QueueStore.create(snapshot);
    render();
  });
}
import React, { Component } from 'react';
import { inject, observer } from 'mobx-react';
import styled from 'styled-components';
import { Redirect } from 'react-router-dom'
import logo from '../static/images/logo.png';
import Info from '../Info';

import { Main, 
    Header 
} from './styles/common-styles';

import { AppContainer } from './styles/AppContainer';

import { 
    StyledTextField, 
    FormContainer, 
    StyledButton 
} from './styles/form';

const LoginMain = Main.extend`
  min-height: 50em;
  align-items: center;
  justify-content: center;
  flex-direction: column;
';

const LoginTextField = StyledTextField.extend`
  width: 35%;
';

const LoginFormContainer = FormContainer.extend`
  align-items: center;
';

const LoginHeader = Header.extend`
  width: 50%;
';

const LoginButton = StyledButton.extend`
  width: 20%;
';

const LogoImage = styled.img`
  width: 15em;
  margin-bottom: 2em;
';

class Login extends Component {

  render() {
    return {
        
    }
username,
setUsername,
password,
setPassword,
submitLogin,
isAuthenticated,
admin,
state,
reqMessage
} = this.props.authenticationStore;
const { from } = this.props.location.state || { from: { pathname: '/' } };
if (!isAuthenticated)
return (

<AppContainer>
  <LoginMain>
    <LogoImage src={{logo}} />
    <LoginHeader>Kirjaudu sisään käyttäjätunnuksestasi</LoginHeader>
    <LoginFormContainer>
      <LoginTextField
        type='text'
        placeholder='Käyttäjätunnus'
        value={username}
        onInput={(e) => setUsername(e)}
      />
      <LoginTextField
        type='password'
        placeholder='Salasana'
        value={password}
        onInput={(e) => setPassword(e)}
      />
      <LoginButton
        type='submit'
        value='Kirjaudu'
        disabled={
          (username.length < 3 && password.length < 3)
          ||
          state === 'error'
        }
        onClick={(e) => {
          e.preventDefault();
          submitLogin();
        }}
      />
    </LoginFormContainer>
  </LoginMain>
  <Info message={reqMessage} color='red' />
</AppContainer>
App Component

```javascript
import React, { Component } from 'react';
import DevTools from 'mobx-react-devtools';
import { observer, inject } from 'mobx-react';
import { AppContainer } from './styles/AppContainer';
import { MainContainer } from './MainContainer';

export default inject(((queueStore) => ({
    queueStore: queueStore
 }))(observer(
    class App extends Component {

        componentDidMount() {
            window.onbeforeunload = () => {
                const list = this.props.queueStore.combinedQueueList;
                const loading = list.some(i => i.state === 'pending')
                if (loading) {
                    console.log('Prevented refresh because some items are loading')
                    return false;
                } else {
                    console.log('Nothing loading so refresh prevention not needed')
                    return;
                }
            }
        }

        render() {
            return (
                <AppContainer>
                    <MainContainer />
                    <DevTools />
                </AppContainer>
            );
        }
    }));
```
import { inject, observer } from 'mobx-react';
import { Switch, Route, Redirect, withRouter } from 'react-router-dom';
import CommandEdit from './CommandEdit';
import FilterContainer from './FilterContainer';
import HistoryTable from './HistoryTable';
import QueueContainer from './QueueContainer';
import HistoryItemView from './HistoryItemView';
import CommandItemView from './CommandItemView';
import Notfound from './Notfound';
import { RouteNotFound } from './Notfound';
import { Main } from './styles/common-styles';
import { MenuBar } from './MenuBar';
import Info from './Info';
import { lightGreen } from './styles/colors';

export const MainContainer =
  withRouter(
    inject((rootStore, ...rest) => {
      queueStore: rootStore.queueStore,
      historyStore: rootStore.historyStore,
      authenticationStore: rootStore.authenticationStore,
      userStore: rootStore.userStore,
      commandItem: rootStore.commandItem,
      fastCommandItem: rootStore.fastCommandItem,
      ...rest
    }))(
      observer(
        ({ queueStore, historyStore, authenticationStore, commandItem, fastCommandItem, ...rest }) => {
          //Combining queueStore lists so that they will be discoverable for React Router from the same list
          const queueContainerList = queueStore.combinedQueueList;
          const { isAuthenticated, reqMessage, state, admin }
            = authenticationStore;

          const { location } = rest;
          if (isAuthenticated)
            return (
              <div>
                <MenuBar />
                <Main>
                  <Switch>
                    { admin &&
                      <Route exact path='/command' component={() => <CommandEdit fast={false} />} />
                    }
                  </Switch>
                </Main>
              </div>
```
function render(route) {
  const view = route.component.view;
  return typeof view !== undefined && view ?
    <View view={view} /> : <RouteNot Found />
}

function findCommandItem(queueId) {
  return typeof queueContainerList !== undefined && queueContainerList.length > 0
    ? <Route component={CommandItem} render={() => ({ match }) => {
        const view = queueContainerList.find((v) => (v.id.toString() === match.params.queueId));
        return typeof view !== undefined && view ?
          <CommandItemView from={[typeof view.successful_commands === 'undefined' ? 'pika' : 'komennot']} view={view} /> : <RouteNotFound />
      }} /> : undefined;
}

function findHistoryItem(historyId) {
  return typeof historyStore.historyList !== undefined && historyStore.historyList.length > 0
    ? <Route component={HistoryItem} render={() => ({ match }) => {
        const view = historyStore.historyList.find((v) => (v.command_id.toString() === match.params.historyId));
        return typeof view !== undefined && view ?
          <HistoryItemView from={[`HISTORIA ID ${view.id}`]} HistoryItem={view} /> : <RouteNotFound />
      }} /> : undefined;
}

function switchRoute(location) {
  if (location.pathname !== '/history/table' &&
      location.pathname !== '/fast' &&
      location.pathname !== '/history') {
    return <Route component={NotFound} />;
  } else if (location.pathname === '/fast') {
    return <Route component={CommandEdit} fast={true} />;
  } else if (location.pathname === '/history') {
    return <Route component={FilterContainer} />;
  } else if (location.pathname === '/history/table') {
    return <Route component={HistoryTable} />;
  } else if (queueContainerList !== undefined && queueContainerList.length > 0) {
    return <Route path='/queue/:queueId' render={() => ({ match }) => {
      const view = queueContainerList.find((v) => (v.id.toString() === match.params.queueId));
      return typeof view !== undefined && view ?
        <CommandItemView from={[typeof view.successful_commands === 'undefined' ? 'pika' : 'komennot']} view={view} /> : <RouteNotFound />
      }} />;
  } else if (historyStore.historyList !== undefined && historyStore.historyList.length > 0) {
    return <Route path='/history/table/:historyId' render={() => ({ match }) => {
      const view = historyStore.historyList.find((v) => (v.command_id.toString() === match.params.historyId));
      return typeof view !== undefined && view ?
        <HistoryItemView from={[`HISTORIA ID ${view.id}`]} HistoryItem={view} /> : <RouteNotFound />
      }} />;
  } else {
    return <Switch> {
      location.pathname !== '/history/table' &&
      <QueueContainer />
    } </Switch>;
  }

  return <Main> {
    <Info message={
      reqMessage ||
      commandItem.message ||
      fastCommandItem.message
    } color={
      state === 'error' ||
    } />
  </Main>;
}
Appendix 25. CommandEdit Component is the Parent Component for CommandEditForm and FastEditForm

```javascript
import React, { Component } from 'react';
import { observer, inject } from 'mobx-react';
import CommandEditForm from './CommandEditForm';
import FastEditForm from './FastEditForm';

export default inject((rootStore) => ({
  authenticationStore: rootStore.authenticationStore,
  queueStore: rootStore.queueStore,
  commandItem: rootStore.commandItem,
  fastCommandItem: rootStore.fastCommandItem,
  userStore: rootStore.userStore
}))(observer(
  ({ authenticationStore, commandItem, fastCommandItem, queueStore, userStore, ...rest }) => (
    !rest.fast ?
      <CommandEditForm
        authenticationStore={authenticationStore}
        commandItem={commandItem}
        queueStore={queueStore}
        userStore={userStore}
      />
    :
      <FastEditForm
        authenticationStore={authenticationStore}
      />
    
```
import React, { Component } from 'react';
import { observer } from 'mobx-react';
import { HeaderBox, LeftContainer } from './styles/common-styles';
import { operationList, reasonList } from '../..//constants';
import { preventSubmitOnEnterKey } from '../..//util/util';
import {
  FormContainer,
  StyledButton,
  StyledTextField,
  StyledSelect,
  ButtonRow
} from '../..//styles/form';
import { StyledFontAwesomeIcon } from '../..//styles/icons';
import { lightGreen, lightGrey } from '../..//styles/colors';
import { LocationList } from '../..//LocationList';

export default observer((
  { authenticationStore, commandItem, queueStore, userStore }) => (

  <LeftContainer>
    <HeaderBox header='KOMENNOT' />
    <FormContainer>
      <div className='command-input-wrapper'>
        <StyledTextField
          type='text'
          placeholder='Syötä käyttöpaikka'
          value={commandItem.location}
          onChange={commandItem.setLocation}
          onKeyPress={preventSubmitOnEnterKey}
        />
        <a onClick={() => commandItem.addLocations(userStore, authenticationStore)}>+
          <StyledFontAwesomeIcon icon='plus-circle'
            color={commandItem.location === '' ? lightGrey : lightGreen}
            size='1.3em'
          />
        </a>
      </div>
    </FormContainer>
  </LeftContainer>
</div>

appendix 26. CommandEditForm
<Select commandItem={commandItem} reasonSelect={true} />

<ButtonRow>
  <StyledButton type='submit' value='Lähetä'
    disabled={
      commandItem.locations.length < 1
      ||
      !commandItem.operation
      ||
      (commandItem.operation.toLowerCase().includes('disconnect')
        &&
        !commandItem.reason)
    }
    onClick={(e) => commandItem.commandSubmit(e, queueStore, userStore, authenticationStore)}
    onKeyPress={preventSubmitOnEnterKey}
  />
  <StyledButton type='reset' value='Tyhjennä'
    disabled={commandItem.locations.length < 1 && !commandItem.operation}
    onClick={() => commandItem.resetCommand()}
  />
</ButtonRow>
</FormContainer>
<LocationList
  locations={commandItem.locations_with_addresses}
  removeLocation={((location, index) => commandItem.removeLocationWithAddress(index))}
/>
</LeftContainer>
});

const Select = observer(((commandItem, reasonSelect = false) => (
  !reasonSelect ?
  <StyledSelect
    value={commandItem.operation} onChange={((e) => commandItem.setOperation(e))}
  >
    <option className='default' value=''>Valitse Komento</option>
    {operationList.map((o, index) => (<option key={index} value={o.en}>{o.fin}</option>))}
  </StyledSelect>

```
Apppendix 27. CommandItemView

```javascript
import React from 'react';
import styled, { css } from 'styled-components';
import { observer } from 'mobx-react';
import { LeftContainer, HeaderBox, DescList } from './styles/common-styles';
import { StyledFontAwesomeIcon } from './styles/icons';
import { lightGreen } from './styles/colors';
import Locations from './Locations';
import { localizedDate } from '../../util/time';
import { localize } from '../../util/util';
import Details from './Details';

const DescWrapper = styled.div
  display: flex;
  align-items: center;

  svg {
    margin-left: auto;
    margin-right: 25%;
  }

  .svg-margin-right {
    margin-right: 24%;
  }

export default observer((from, CommandItem) => {
  const {
    locations,
    location_to_submit,
    successful_commands,
    failed_commands,
    operation,
    creation_time,
    request_duration,
    ending_time,
```
reason,
info,
state
} = CommandItem;

let header = state === 'pending' || !ending_time ? localizedDate(creation_time) :
  `$(localizedDate(creation_time)) - $(ending_time)`

return (  
  <LeftContainer>
    <HeaderBox
      header={header}
      backTo={from.includes('komennot') ? '/command' : '/fast'} />
    <DescWrapper>
      <DescList>
        <li>
          <span className='type'>Toiminto: </span><span>{localize(operation)}</span>
        </li>
        { reason &&
          <li>
            <span className='type'>Syy: </span><span>{localize(reason)}</span>
          </li>
        }
        { locations &&
          <li>
            <span className='type'>Yhteismäärä: </span>{locations.length > 1 ? 'Käyttöpaikka: ' : ''}
          </li>
        }
        { location_to_submit &&
          <li>
            <span className='type'>Käyttöpaikka: </span><span>{location_to_submit}</span>
          </li>
        }
        { request_duration &&
          <li>
        }
<span className='type'>Kokonaiskesto: </span><span>{request_duration}</span></li>
</DescList>
{
state === 'error' ?
<StyledFontAwesomeIcon
    icon='times'
    color='red'
    size='3em'
/>
:
state === 'pending' ?
<StyledFontAwesomeIcon
    className='svg-margin-right'
    icon='cog'
    color='white'
    size='3em'
    spin={true}
/>
:
<StyledFontAwesomeIcon
    className='svg-margin-right'
    icon='check'
    color={lightGreen}
    size='3em'
/>
}
</DescWrapper>
{
info &&
<Details {...info} />
}
{
successful_commands && failed_commands &&
<Locations
    state={state}
    successful_commands={successful_commands}
    failed_commands={failed_commands}
/>

</LeftContainer>
)
});

Appendix 28. FastEditForm

import React from 'react';
import { observer } from 'mobx-react';
```javascript
import { HeaderBox, LeftContainer, DescBox } from './styles/common-styles';
import { operationList } from './../../../constants';
import { preventSubmitOnEnterKey } from './../../../util/util';
import { StyledFontAwesomeIcon } from './styles/icons';
import { FormContainer, StyledButton, StyledTextField, ButtonRow, LocationToSubmit } from './styles/form';
import { lightGreen, lightGrey } from './styles/colors';

export default observer(((fastCommandItem, queueStore, userStore, authenticationStore)) => {

  <LeftContainer>
    <HeaderBox header='PIKASUORALUENTA' />
    <DescBox>
      <span className='desc'>Toiminto: </span><span>{operationList[0].fin}</span></DescBox>
    </LeftContainer>
    <FormContainer>
      <div className='command-input-wrapper'>
        <StyledTextField
          type='text'
          placeholder='Syötä käyttöpaikka'
          value={fastCommandItem.location}
          onInput={(e) => fastCommandItem.setLocation(e)}
          onKeyPress={(e) => fastCommandItem.AddLocationOnEnter(e)}
        />
        <a onClick={() => fastCommandItem.addLocationToSubmit()}>}
          <StyledFontAwesomeIcon
            icon='plus-circle'
            color={fastCommandItem.location === '' ? lightGrey : lightGreen}
            size='1.3em'
          />
          <StyledButton type='reset'
```
```
value='Tyhjennä'
disabled={
  !fastCommandItem.location
  &&
  !fastCommandItem.location_to_submit
}
onClick={() => fastCommandItem.resetFastCommand()}
</ButtonRow>
</FormContainer>
<LocationToSubmit>
<span onClick={() => fastCommandItem.removeLocationToSubmit()}>
  {fastCommandItem.location_to_submit}
  <StyledFontAwesomeIcon
    className='show-delete'
    icon='times-circle'
    color='red'
  />
</span>
</LocationToSubmit>
</LeftContainer>

Appendix 29. Util function initStore to initialize the store from sessionStorage

```javascript
import { onSnapshot } from 'mobx-state-tree';

export const initStore = (initObject, store, storageType) => {
  const json =
    window.sessionStorage.getItem(`${storageType}`) || false ?
      JSON.parse(window.sessionStorage.getItem(`${storageType}`)) :
      initObject;

  if (json && store.is(json)) {
    initObject = json;
  }
  return initObject;
};
```