Lotta Laukkanen

Front end development with ClojureScript framework Re-frame

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

1 April 2021

Abstract

Author:	Lotta Laukkanen
Title:	Front end development with ClojureScript framework Re-frame
Number of Pages:	36 pages + 3 appendices
Date:	1.4.2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Media Technology
Instructors:	Tero Kojo, Manager Software Engineering Ulla Sederlöf, Senior Lecturer

---

This thesis addresses the difference between popular web development framework ReactJS and less known Re-frame, a framework based on React, and compares their strengths and weaknesses in front end development. Due to the nature of Re-frame, this thesis also explores reactive functional programming and the primary languages of these frameworks: JavaScript and ClojureScript. The topic is addressed from an angle of adopting Re-frame and Clojure into enterprise use as an alternative to ReactJS.

A web-based user interface for a laboratory robot is used as a use case for the technology. One workflow wizard is in the center of the project. Wizards are used to create pipetting workflows by setting parameters and creating plate maps for the robot. The use case presents the most relevant features of the framework in the context of front-end development in a way approachable for people not familiar with the technology. It was discovered that despite Clojure's unpopularity in front end development, ClojureScript paired with Re-frame is suitable for the purpose. The project was carried out for an international laboratory equipment manufacturer company.

Keywords: Clojure, React, Re-frame, user interfaces

Tiivistelmä

Tekijä:	Lotta Laukkanen
Otsikko:	Käyttöliittymäkehitys ClojureScript-kirjasto Re-framella
Sivumäärä:	36 sivua + 3 liitettä
Aika:	1.4.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mediatekniikka
Ohjaajat:	Manager Software Engineering Tero Kojo Lehtori Ulla Sederlöf

Opinnäytetyössä perehdyttiin suositun web-sovelluskehitysteknologia ReactJS:n ja vähemmän tunnetun React-pohjaisen Re-framen eroihin ja punnittiin niiden heikkouksia ja vahvuuksia käyttöliittymäohjelmoinnissa. Re-framen luonteen vuoksi opinnäytetyössä pureuduttiin myös reaktiivisen funktionaalisen ohjelmoinnin konseptiin sekä mainittujen kirjastojen pääasiallisten kielten, JavaScriptin ja ClojureScriptin, eroavaisuuksiin. Käsittelyn näkökulmaksi otettiin Re-framen ja ClojureScriptin ottaminen yrityskäyttöön tilanteessa, jossa sitä punnitaan vaihtoehtona ReactJS:lle.

Opinnäytetyössä käytettiin esimerkkinä laboratoriorobotille toteutettavaa web-pohjaista käyttöliittymää ja keskityttiin erityisesti kehittämään yhtä käyttöliittymän osakokonaisuutta, jolla luodaan ohjatusti uusia työnkulkuja syöttämällä sovellukseen työnkulun parametreja ja kuoppalevykaavio. Esimerkkityön kautta tarkasteltiin teknologian olennaisimpia piirteitä käyttöliittymäkehityksessä ja pyrittiin avaamaan Re-framella toteutettavia projekteja myös yleisellä tasolla.

Opinnäytetyössä ilmeni, että vaikka Clojure ei ole kovin suosittu kieli käyttöliittymäkehityksessä, Re-frame-kirjaston kanssa ClojureScript soveltuu tähän takoitukseen hyvin. Työ toteutettiin kansainväliselle laboratoriolaitteita valmistavalle tekniikan alan yritykselle.

Avainsanat: Clojure, React, Re-frame, käyttöliittymät

Contents

List of Abbreviations

JS:	JavaScript
CLJ:	Clojure
CLJS:	ClojureScript
SPA:	Single Page App
UI:	User Interface
GUI:	Graphical User Interface
LISP:	List Processing Language
DOM:	Document Object Model
FRP:	Functional Reactive Programming
API:	Application Programming Interface
HTML:	Hypertext markup language
CSS:	Cascading Style Sheets
JSX:	JavaScript XML
IDE:	Integrated Development Environment
REPL:	Read-evaluate-print loop
TCA-kit:	The Human T Cell Activation Cell and Cytokine Profiling Kit

1 Introduction

This thesis introduces a ClojureScript framework Re-frame and compares it to ReactJS, a popular web development framework. Along with Re-frame a programming language called ClojureScript is introduced and compared to JavaScript, the language primarily used with ReactJS. This thesis also briefly displays the development of a Single Page App (SPA) UI created using Re-frame.

The project done within the scope of this thesis is a fraction of a larger project. The larger project strives to create a web based UI for a laboratory robot. The UI will be used to create liquid handling workflows and to run them in simulations or with the robot. The technologies for this UI were chosen by the project's Tech Lead prior to the beginning of the thesis project. Because the chosen language and framework are new to the company, this thesis addresses the pros and cons of the choices and introduces the basics. The thesis project focuses on the basics of the technology using the TCA-kit workflow wizard as a case example, since it is a simple entity with all the crucial Re-frame functionalities.

Clojure is a functional LISP programming language originally released in 2007(1; 2). As a functional language, it features a strong set of immutable data structures and favours pure functions. ClojureScript, first released in 2011 (3), is essentially a Clojure compiler targeting JavaScript. ClojureScript projects also tap into Google Closure Library and Closure Compiler, which offer a wide range of tools for UI development and DOM manipulation while optimizing the code. While compilers usually compile programming language into machine code, Google Closure compiler compiles JavaScript into optimized JavaScript instead. (4; 5; 6).

Using ClojureScript it is possible to create web and mobile applications with the same flexibility and toolset, as one would have with JavaScript, while avoiding

difficulties related to typical JavaScript development. This is helpful especially in large-scale projects.

The framework used in the thesis project is Re-frame. It is a single framework for web application development. Re-frame uses ClojureScript-React interface, Reagent, and adds Redux-like state management to it.

2 UI development with ClojureScript

2.1 Functional programming and user interfaces

Functional programming is a style of programming based on pure functions - ones that only take data as parameters and output data without having “side effects”. Functional programming avoids mutable state and uses immutable data structures (7, chapter 1.3).

As can be seen from the definition above, functional programming does not sound compatible with any application that should have a state the user can change - with any user interface, effectively. This is where Functional Reactive Programming (FRP) comes into play.

Reactive programming is a broad term referring to event-based programming style, which responds to input and can be viewed as a flow of data as opposed to flow of control. It does not define any specific means to achieve these goals and they can be met with several different technologies and styles. Functional reactive programming is a form of reactive programming that strives for functionality as far as it is possible in the scope of reactiveness. (7, chapter 1.3).

While achieving usage of pure functions in an app can be implemented almost perfectly, use of shared mutable state is a necessity for an app that has to make updates on the client side without updating the entire website every time the user makes a change. In the case introduced here this is handled by Re-frame. The framework manages the state of the app in a similar manner as Context API or Redux would do in a React app. In a way, functions using data from this

state could be considered impure and thus the goal of pure functions can be achieved almost completely, but not quite (8).

In addition to the improvements in the state management and functions, functional reactive programming seeks to solve the problem of observer pattern. The dominant way of propagating events in software these days are listeners or callbacks (7, chapter 1.7). While being widely used they also have several issues related to the event listeners receiving the events in an awkward order or lacking information of what to update. This can cause, for example, missing the first event, threading and leaking of callbacks and issues with the state, or UI updates happening in unintended ways. (7, chapter 1.7; 9.) With FRP principles, one should be able to avoid these pitfalls and achieve a modular and easy-to-test software. Using powerful functional programming with immutable data in JS can be done with additional libraries, but it is in-built in Clojure language and thus is native to Re-frame.

2.2 React and Reagent

To understand what Reagent and Re-frame do, it is necessary to understand the basics of React as a framework. React is a UI development library used to build Single Page Apps for the web. It mainly uses JavaScript, but some HTML and CSS is also needed. With React it is possible to create dynamically updating web content and reusable components for the application. Originally, React used JavaScript classes to create the components, but lately functional components have gained popularity due to the advantages of functional reactive programming.

Reagent is a ClojureScript interface to React. It was created to enable powerful functional programming with React before functional components were released (8). That is, it is based on React class components and does not fully support all features of functional components (20). Reagent uses Hiccup data structure, explained in detail in chapter 3 (3.1.2 Components), to create HTML by using nested Clojure vectors as HTML elements (10). It makes it possible to write UI

code almost only with ClojureScript functions. Reagent components are essentially pure or almost pure ClojureScript functions that take parameters as basic Clojure data types. Reagent components re-render only when their data changes, which together with ClojureScript optimizing enhances the software performance to a point where one rarely has to think about it. For this reason, even the simplest components should be called as Reagent components instead of Clojure functions, even if calling them as Clojure functions would be possible. (11.) Holding on to component syntax at all times is also a good practice for code uniformity and makes the code clearer to read.

2.3 Re-frame

Re-frame is a data-oriented framework for ClojureScript. It uses Reagent as an interface between React and ClojureScript and adds more data and state handling that can be compared to, for example, Redux or Elm (12).

Re-frame handles the data loop by following these steps:

1. Event dispatch

– Events are dispatched when something changes, for example on a UI event.

2. Event handling

– Declarative descriptions of a needed effect are computed.

3. Effect handling

– The effect declared in step 2 is executed. If it makes changes to the application state, the following steps unravel as well.

4. Query

– Extraction of the changed data from the app state.

5. View

– ViewFunctions i.e. Reagent components describe DOM elements in Hiccup syntax.

6. DOM

– DOM nodes provided in step 5 are actioned by Reagent.

The data flow of Re-frame can be illustrated with the water cycle diagram (Figure 1). The diagram can be interpreted as a depiction of the data loop going through the six steps of iteration to keep data fresh, as humorously implied in the Re-frame documentation. (13.)

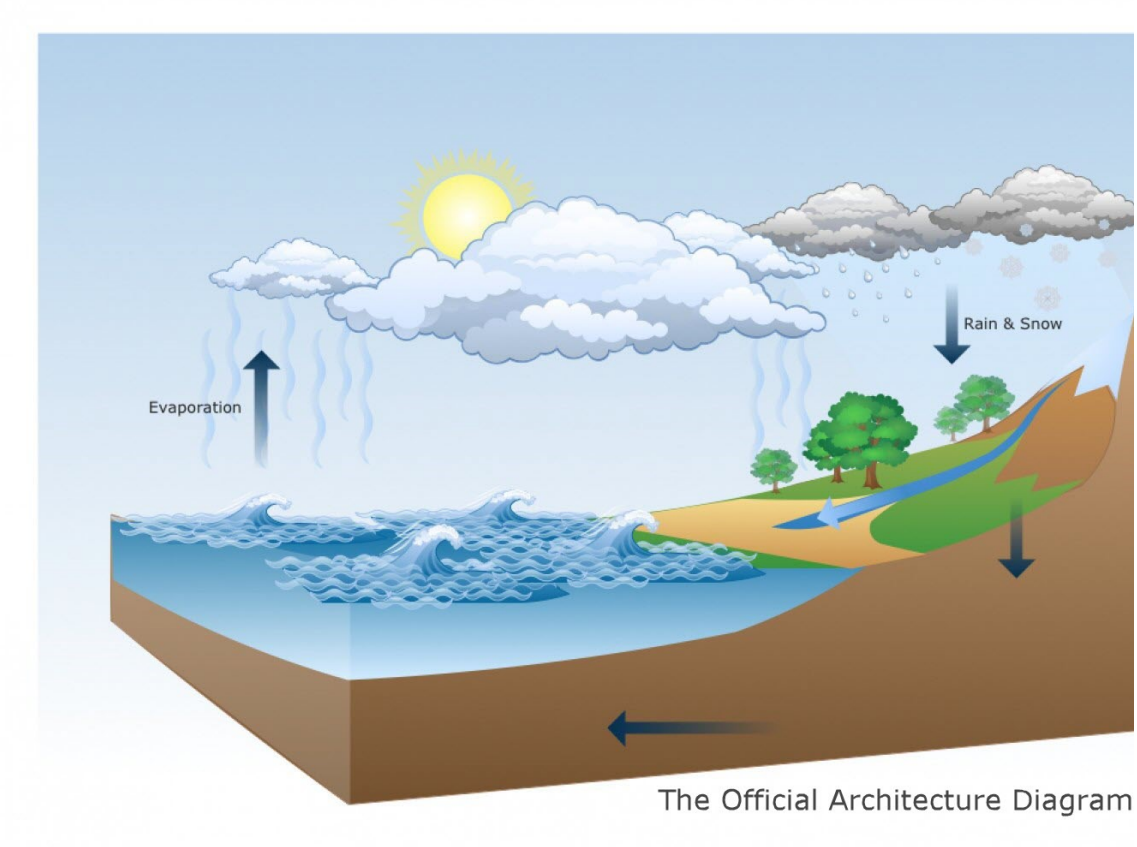

Figure 1: A water cycle diagram symbolising the steps and the smoothness of Re-frame data loop (13).

Re-frame stores the entire application state into one place named app-db, which is similar to an actual database. In practice, app-db is a Reagent atom containing the application state as a map. (14.) Atom is a reference type in Clojure and will be described in more detail in chapter 3 “Comparing Re-frame and React”. Using a Reagent atom for this means that components using the atom get re-rendered whenever the value changes (11).

The handling of data in a Re-frame app can also be depicted as The Signal Graph (Table 1). Simply, a needed piece of data is extracted from app-db, computed into derived data - however, this step is sometimes left out in more simple solutions - and then *subscribed* to be used in DOM. (15.)

Table 1. Table created following the logic of an example in the Re-frame documentation (15).

Layer name	Data	Excel function equivalent	Description
app-db	Bruce | Wayne | 1007 Mountain Drive | Gotham		Layer 1: app-db holding app state
extraxtion	Bruce	=LEFT(B1,FIND(" | ",B1))	Layer 2: Extracting piece of data
computation	Hello Bruce	=CONCAT("Hello ",B2)	Layer 3: Computing the needed value for the view
view	[:div "Hello Bruce "]	=CONCATENATE("[:div """ ,B3,"""]")	Layer 4: Computing Hiccup

However, these steps or nodes are created in the opposing order: changes in the view create a *subscribe*, which in turn launches the data extraction from the app-db. The opposing flows can be seen in the diagram below (Figure 2). (15.)

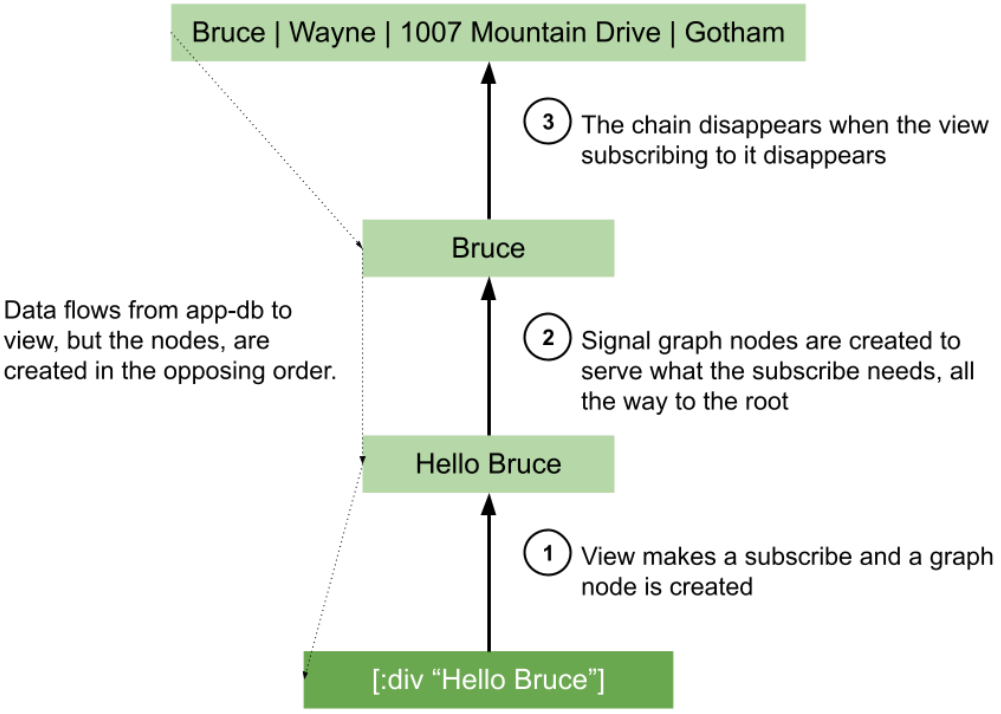

Figure 2: Signal graph describing how the Signal Graph nodes are created (15).

3 Comparing Re-frame and React

3.1 Technical differences

Despite the fact that Re-frame is technically one way to use React, these two technologies are very different. Re-frame consists of Reagent, React-interface, and state management. Even though this could be and often is compared to the combination of React and Redux, React in fact has an in-built global state management tool called Context API. That is why the separate library Redux is mentioned, but not considered as the default state management option for React in this thesis. This chapter addresses five areas that greatly differ between Re-frame and React.

3.1.1 Syntax

Syntax is the very first thing a person sees when looking at code. The language used with Re-frame is Clojure. Clojure syntax looks very different from that of JavaScript for a couple of reasons. Firstly, Clojure is homoiconic, meaning that the data and code are essentially the same; consequently, code is also presented as data structures. As typical for a LISP Clojure uses a lot of parentheses (16), although square and curly brackets are also used regularly. When compared to Python Clojure parentheses seem excessive, but when compared to JavaScript they are in fact quite equal. The number of brackets and parentheses might still seem high because Clojure syntax is more compact than JavaScript. This can create the illusion of more parens, when in fact there are just fewer lines of code representing the same functions with roughly the same amount of parentheses. The following code snippets demonstrate how the same function in these languages can have different type of brackets and different numbers of lines, while maintaining a similar number of brackets.

The following code (Listing 1) written in JavaScript has eight pairs of brackets altogether, half of them are round and half of them are curly brackets. This function takes ten to twelve lines.

```
function buildString(someInteger) {
      var question = "Initial text";
      if (someInteger == 1) {
            question += " put this string ";
      } else if(someInteger == 2) {
            question += " oh! another string ";
      } else if(someInteger == 3) {
            question += " guess what? ";
      }
      return question;
}
```

Listing 1. A JavaScript function that builds a string (17).

In the following Clojure snippet (Listing 2) the same is done in six lines, using eleven pairs of brackets. Two pairs are square brackets and the rest are the round parentheses LISP languages are notorious for. The difference in parens per line from JavaScript to Clojure, however, is three pairs of parens more and

up to six lines of code less. Thus the paren to code ratio may seem intimidating even though the difference in parens per function is not that massive.

```
(defn build-string [some-integer]
     (let [question "Initial text; "]
          (cond
               (= some-integer 1) (str question "Add string one")
               (= some-integer 2) (str question "Add string two")
               (= some-integer 3) (str question "Add string three"))))
```

Listing 2. A Clojure function that builds a string (17).

These examples also show another major difference between the languages: the way functions are assembled. When in JS function parameters are declared inside parentheses right after the function call, in Clojure functions are handled as lists, with the first element representing the verb and the rest representing parameters. This difference becomes especially visible with arithmetic operations: the operator is placed in the beginning of the function. The following snippet (Listing 3) returns value 2021.

```
(+ 20 2000 1)
```

Listing 3. A basic arithmetic function in Clojure.

In JS the arithmetic operators are infixed which could be argued to be more intuitive to an average user than the Clojure solution. This is due to the way mathematics is usually taught: the snippet above written in JS or in a math textbook would be (20 + 2000 + 1). Handling the arithmetic operators, the same way as other function calls, however, makes the language syntax uniform. (18)

When discussing React and Re-frame, components are in a very central role. While the components themselves will be addressed in the following section, their syntax is a noteworthy detail. While React often utilizes a syntax extension called JSX to describe UI (19), Re-frame does the same using Clojure vectors. This data structure describing HTML is known as Hiccup (10). Reagent adds some extensions to usual Hiccup, but the markup style is the same (20).

3.1.2 Components

UI elements in Re-frame are described with nested Hiccup-style vectors that follow these general rules: first element is keyword or symbol, second one is a map that represents the attributes of the element and the rest are either vectors representing child elements or string literals representing child text nodes. If the first element is a keyword, the vector is considered an HTML element. If it is a symbol, the vector is considered a component. Behind the scenes Reagent uses React function React.createElement to render the UI elements. (10.)

React components can be either class components or functional components. Since functional components are a closer equivalent to Clojure vectors, only they are addressed here. A functional React component is essentially a function which returns a UI element in some form. As mentioned before, it is often recommended to use JSX to describe what kind of UI a React component should render. JSX is a syntax extension that makes it possible to have the markup and the logic of the app in the same file. JSX syntax resembles HTML code, providing HTML tags and hierarchies for UI development in JavaScript. It is also possible to write JavaScript code inside JSX, (19.) however, JSX is not a requirement for using React, since JSX is just a wrapping for React.createElement. In other words, React components can be created directly by using that function (21), which Reagent also utilizes behind the scenes.

The following snippets (Listing 4; Listing 5) show the same styled element in Hiccup and in JSX. The common elements are visible, despite being written differently. The similarity is more clear when looking at the React.createElement function (Listing 6) below them: it is undeniably JS, but the function takes exactly the same parameters in the same order as the Hiccup style element.

```
[:div {:style {:display "grid" :grid-template-columns "auto 1fr"
               :align-items "center" :grid-column-gap "32px"
               :grid-auto-rows "60px"}}
     [:span.with-info "Mix by pipetting"
          [components/infobox "Gently pipette up and down at least 6 times to
completely mix the solution."]]]
```

Listing 4. A UI element presented with Hiccup syntax.

```
<div style={{
        display: "grid",
        gridTemplateColumns: "auto 1fr",
        alignItems: "center",
        gridColumnGap: "32px",
        gridAutoRows: "60px"}}>
        <span className="with-info">
          Mix by pipetting
          <Infobox>
            Gently pipette up and down at least 6 times to completely mix the
solution.
          </Infobox>
        </span>
      </div>
```

Listing 5. The same element as above (Listing 4) in JSX.

```
React.createElement("div", {
      style: {
        display: "grid",
        gridTemplateColumns: "auto 1fr",
        alignItems: "center",
        gridColumnGap: "32px",
        gridAutoRows: "60px"
      }
    }, React.createElement("span", {
      className: "with-info"
    }, "Mix by pipetting", React.createElement(Infobox, {
      prop: "Gently pipette up and down at least 6 times to completely mix the
solution."
    })));
```

Listing 6. A React element created with a function.

React elements are essentially objects created with the createElement function. The following code snippet (Listing 7) shows the simplified form of such an object. If the elements were written directly in these data structures, they could be manipulated by merging or concatenating with normal JS operations. These operations cannot be applied to JSX elements, but since Hiccup is Clojure data structure, similar Clojure operations work on Hiccup elements.

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

Listing 7. A sample React element object (19).

3.1.3 Development environment and tooling

There are several options for IDE that work with Re-frame and React. Technically both frameworks can be edited with a great variety of IDEs, but it is highly recommended to choose one that highlights paired parentheses or has an extension that does so when working with LISP languages due to the amount these languages use parentheses. The one picked for the project described in this thesis is the IDE IntelliJ paired with the Cursive extension. According to the Popularity of Programming Language Index in March 2021, IntelliJ was the sixth most searched IDE and statistics provided by G2 it is within the top 3 most popular IDEs (22; 23). A more popular IDE Visual Studio Code (22; 23) has an extension for Clojure and ClojureScript development too, which makes it a good alternative for Re-frame development. React can be written directly into HTML files for learning purposes, but for production use a Node.js environment needs to be installed (24). Because React needs Node for production versions, Re-frame that uses this functionality requires installation of Node too to compile builds for deployment.

Re-frame and React both use a terminal tool to automate and build projects, and to manage dependencies. With React, recommended tools are npm, which comes along with Node.js (25), or yarn. With Re-frame Leiningen is the recommended tool since it is specifically designed for Clojure and ClojureScript. (26.) There are beginner-friendly starting points available for both technologies utilized with aforementioned management tools. React documentation recommends creating a new project using the Create React App environment, which initializes the React project folders with the required files and folders. Additional tools and libraries can be added on the go after initialisation via

terminal. The client-only template recommended in Re-frame documentation adds shadow-cljs and cljs-devtools and creates the required folders and files for a new project. The mentioned tools enable the use of some developer tools like Chrome devtools, live reload of the app and REPL (Read-Evaluate-Print-Loop), a terminal-like interactive code editor. Additional tools and libraries can be added via the command line after creating the project. Back end and full stack templates are separate and are set up with different commands. (26.)

3.1.4 State management

In both React and Re-frame it is possible to store app state somewhere that could be considered to exist outside the app itself. In Re-frame the state is managed using Reagent atoms. The data flows in a data loop and the app state, named app-db, is never mutated but swapped into an updated version whenever the state should change. (13; 14.) This is how Redux works too: events get dispatched and then trigger a chain of events that leads to changes in the app state. Redux will not make changes to the Redux store either, but rather replaces all data once the updated version of the store is compiled. (27.) Unlike atoms in Re-frame, Redux is a library that is not automatically included in React apps. However, the principle is similar to Re-frame state management and therefore is often compared to it.

The default form of out-of-component state management in modern React are contexts. The official Context API was released within React 16.3 in 2018 (28) and is automatically included in 16.3+ React apps. The intent is to dodge the need to drill component props through generations of child components by making it possible to serve app state from contexts spanning the entire app if needed. Updating these state values happens with set-style functions that replace the data instead of changing it. Context API differs from Redux by, for one, being native to React and requiring less operations for changing the stored values, making it arguably easier to approach for beginners.

3.1.5 Libraries

For ClojureScript there are several libraries available, of which some, but very likely not all, are listed in ClojureScript documentation (29). Most importantly though, ClojureScript can use Google's massive Closure library which serves as the base library of several Google products like Google Maps, Google Books and Gmail (5). Since ClojureScript is compiled with Google Closure compiler the library size is not an issue; only the parts that are used are included in the compilation (3). It is also possible to use npm libraries like Material-UI with Re-frame. The npm ecosystem consists of over one million libraries for different purposes (30). Of course, the aforementioned Google Closure library is included in it and is available for React too. It should also be mentioned that since React is Facebook's creation it does have some very carefully tested and professionally maintained Facebook-originated libraries available. A big issue with importing and using libraries in React apps, however, is the size problem that can be dodged if the code is Closure compiler compatible. Without efficient compiling a React app may easily become bloated due to mostly unused libraries clinging to the app (3).

3.2 Arguments for and against ClojureScript

> "Clojure is arguably simpler, more powerful and more robust than JS" – Rich Hickey, NYC Clojure 20.1.2011

The most important feature of JavaScript is its reach. JavaScript is used in many areas and platforms, and in some places, it is the only possible solution since especially web and mobile platforms favour JS. (3.) Using ClojureScript instead of JavaScript does not mean abandoning JavaScript entirely, but producing JS source code by writing ClojureScript, much like one would do with for example TypeScript. There are, however, arguments for and against the use of ClojureScript instead of JavaScript in programming.

3.2.1 Performance

Because of Google Closure compatible code, apps written in ClojureScript tend to be more efficient concerning the size of the codebase and unused libraries than ones in JavaScript (3). In addition to this, using immutable data structures gives ClojureScript a great advantage when it comes to comparison of data. The equality comparison of immutable data structures is faster than comparison of mutable ones since shallow comparison is enough to determine if there has been changes (31). This is extremely relevant in use cases where information about changes in UI input data is needed to render or re-render UI components correctly, namely in React and in frameworks based on React. Use of immutable data structures removes the need for deep equality comparisons because whenever the data changes, so does the reference.

3.2.2 Scaling and modularity

Since pure functions do not have side effects and they only take in data and then return data, they are easily reusable. Both React and Re-frame have this advantage, although React alone does not enforce the use of modular pure functions the way Re-frame and ClojureScript do. Both also scale well for large applications for different platforms, as can be seen from companies that have used them in their products: React was created by Facebook and ClojureScript in some form is used by at least Twitter, Netflix and Heroku (32). Re-frame, however, has the upper hand of optimized code through Closure compatible output. Simply put, applications compiled with Google Closure take less disk space and therefore have shorter download times than uncompiled ones. In the case of very large online applications that might become crucial.

3.2.3 Error handling

This problem in Clojure has to be mentioned, because it is considered a regrettable design mistake by Rich Hickey himself, the creator of Clojure and ClojureScript (33, page 36). The error messages come from the compiler and

several macros, and Java stack traces indicate that one needs to know or learn Java before being able to effectively use Clojure (33, page 35). This is misleading and confusing.

3.2.4 Popularity of the technology

When choosing technologies for a larger scale development project, workforce is an important thing to consider. This would be a heavy argument against the use of ClojureScript: as can be seen from the diagram below (Figure 3). Based on internet communities in Reddit and LinkedIn, Clojure users are scarce and ClojureScript professionals even fewer, especially compared to the users of JavaScript.

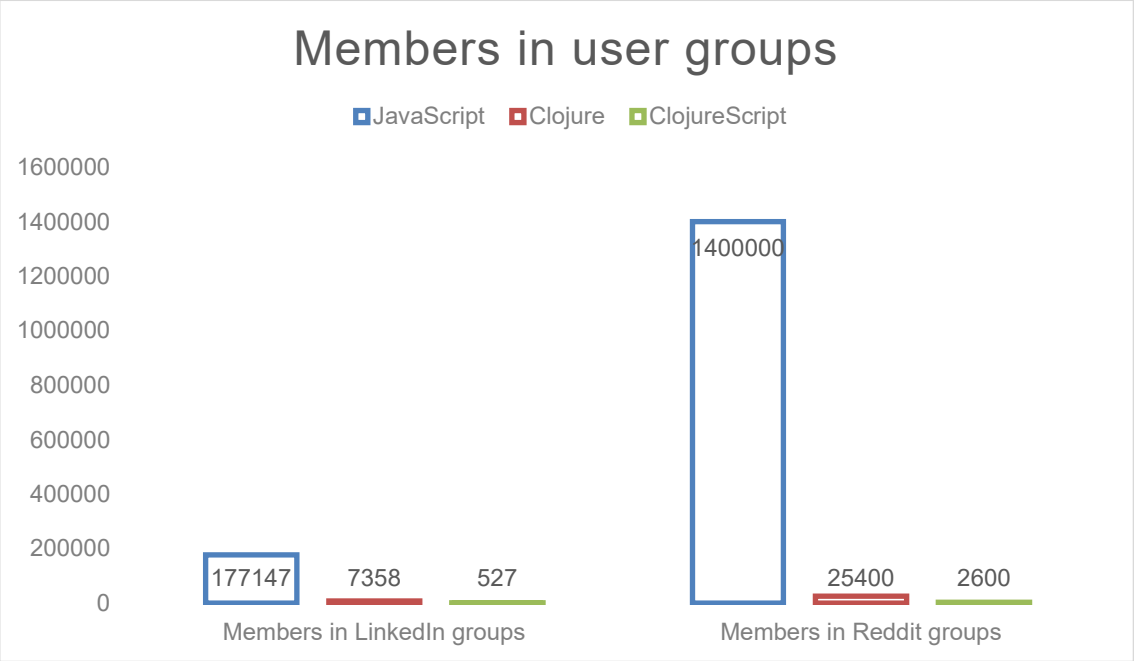

Figure 3: Members in language user groups. Numbers collected from LinkedIn and Reddit 17.2.2021

This might make recruiting developers on the project challenging. Neither language appears to offer very many job openings either, at least via LinkedIn (Figure 4).

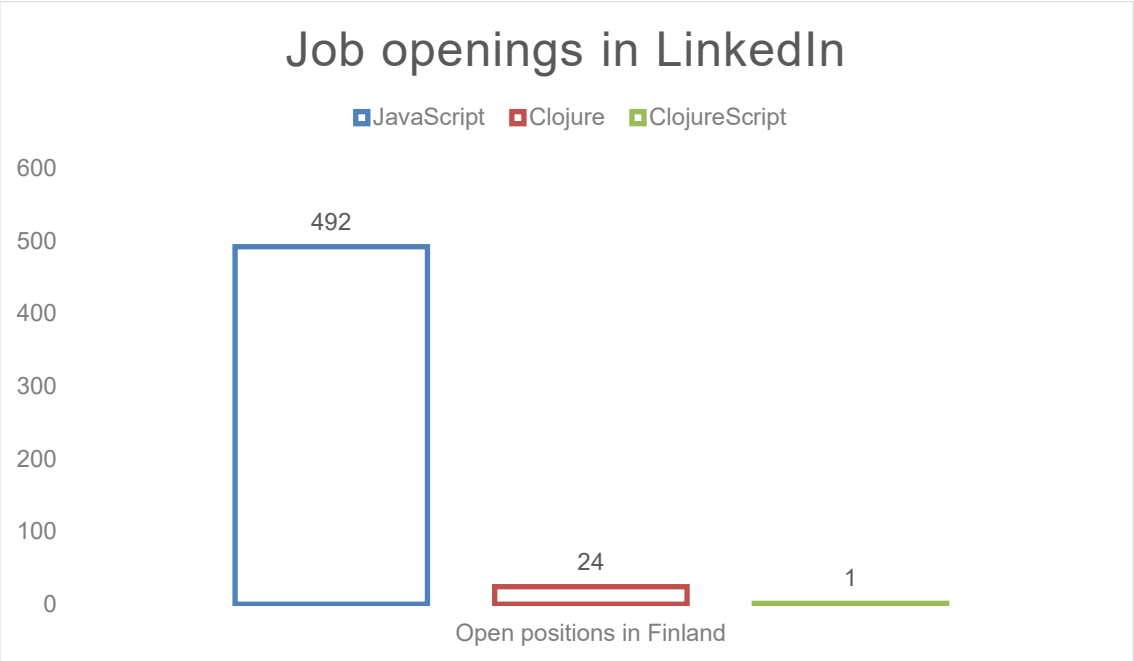

Figure 4: Job openings available in LinkedIn job search 17.2.2021

Another downside of a small user pool is inevitable scarcity of open source material and advice. The following diagram (Figure 5) shows GitHub repository results and StackOverflow questions tagged with these languages. GitHub is an online code-hosting platform currently hosting over 100 million repositories for over 56 million developers (34). StackOverflow is a question and answer site for programmers with over 100 million visitors per month (35).

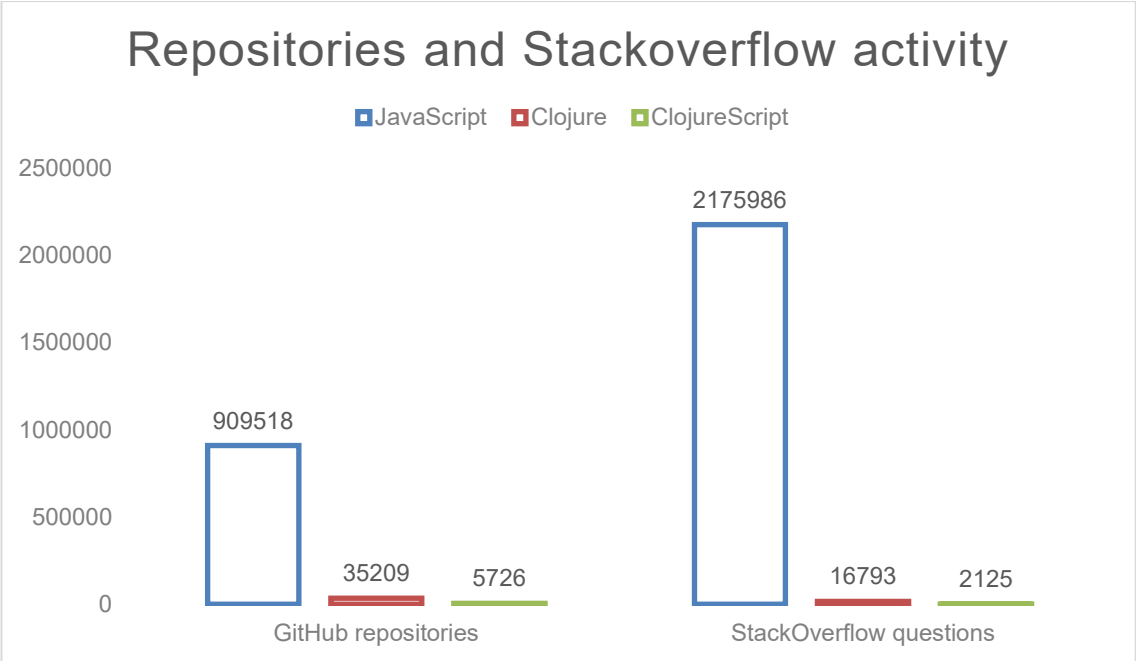

Figure 5: Community content for Clojure and ClojureScript compared to JavaScript. Numbers collected from GitHub and StackOverflow 25.2.2021.

React is a library developed by Facebook. It was created in 2011 and open sourced in 2013. Currently Facebook uses over 50,000 React components in production and therefore has interest in keeping the technology stable and usable for all major browsers, in several versions. As a corporate-supported library, React is also one of the few open source libraries with full-time staff working on it. (36.) This supports Re-frame as well, even though all the React improvements are not available in Re-frame: for example, React hooks that were introduced in React version 16.8 (37) are primarily used in functional components. Reagent, however, requires access to the React component state, which makes it difficult to use functional components (20).

4 Proof of concept: development of a web based UI with ClojureScript

4.1 Project goal and description

The goal of this project was to design, prototype and implement a workflow wizard for a laboratory robot UI with a technology that is very new to the company. The workflow in question is based on the IntelliCyt TCA-kit experiment equipment kit and its manual manufactured by IntelliCyt Corporation (38). The goal was to create a workflow wizard that takes all needed parameters as input and produces a data package to be used for controlling the laboratory robot in a lower level.

The UI is web-based and will be served from the cloud but was not released within the project scope. A local Windows machine was used for development, although the app was originally designed to be developed and built on a Linux platform and the README file contains instructions for setting up the development environment in a local Docker container. The UI itself was written in ClojureScript using the Re-frame framework.

4.2 Starting point

The UI introduced in this thesis had been planned and prototyped by a Customer Experience Designer in autumn 2020 and the development was started by the project's new Tech Lead a few weeks prior to the beginning of the thesis project. Therefore, the development environment and base structure of the app had already been established in the beginning of the project and there was already some programming done for the Graphical User Interface (GUI).

The folder structure and logic hold a trace of the robot interaction in the form of e.g. robot movement coordinate data in folders named *project*, even though this is not evident in the scope of the thesis project. The relevant parts of the folder structure are illustrated below (Listing 8). It is important to notice how the codebase is divided between different types of Clojure used in the project and

by the usage of Re-frame framework in the code. Folders “clj”, “cljc” and “cljs” are divided based on if they can be run with Java only (*clj*) or ClojureScript only (*cljs*) or if they can be used with both (*cljc*). This is because functions borrowed from JavaScript do not work with Java. The parts of the projects that utilize Re-frame are in folders labelled *project_gui* and the ones not using it are in folders named *project*.

```
project_gui                                 Root folder
|
├──.clj-kondo                               Development tool configuration
├──.idea                                    Project settings, IntelliJ specific
├──.lsp
├──.shadow-cljs                             Development tool configuration
├──dev
|     └──cljs
|           └──user.cljs                    Default namespace when working with
REPL
├──devenv                                   Cmd and shell commands for
development environment
├──node_modules                             Node modules
├──resources
|     └──public                             Target location for compiled code to
be deployed
├──src
|     ├──clj                                Java-Clojure
|     |     └──project_gui
|     |           └──css.clj
|     ├──cljc                               Common Clojure for Java and
ClojureScript
|     |     ├──project                      General functions and application
logic
|     |     |     ├──deck.cljc
|     |     |     └──workflow.cljc
|     |     └──project_gui
|     |           └──style.cljc             General style details, like company
colors, with keywords
|     └──cljs                               ClojureScript only
|           ├──project
|           |     └──plate cljc
|           └──project_gui                  GUI
|                 ├──tests                  These tests can be run anytime.
|                 ├──workflow_edit
|                 |     ├──core.cljs        Wizard page initialization and
related scripts
|                 |     ├──tca_details.cljs Single page components
|                 |     ├──tca_plates.cljs
|                 |     └──tca_preview.cljs
|                 ├──components.cljs        Collection of smaller components
|                 ├──core.cljs              Initialization of the GUI
|                 ├──icons.cljs             Icons as vectors in string form
|                 └──material_ui.cljs       Imports from Material-UI library
├──target                                   Temporary compilation folder
├──.gitignore
├──watch.sh                                 Helpful shell commands
├──deploy.sh
├──karma.conf.js                            Testing tool configuration
├──package.json
├──package-lock.json
├──project.clj
├──README.md
├──shadow-cljs.edn                          Development tool configuration
└──project-gui.iml                          Project settings, IntelliJ specific
```

Listing 8. The relevant parts of the project directory tree.

The following diagram (Figure 6) shows the relations between the UI components in this application. Contents of TCA-kit workflow pages, created during the thesis project, are shown in a separate diagram (Figure 7) in more detail. Components with white background have their own namespaces which

contain only the component in question and are not reused; otherwise the namespaces have been color-coded to visualize how components are reused. The components with the same background colour are defined in the same namespace or are re-used from the same namespace.

The diagram does not visualize possible wrapper elements or Material UI components. Neither does it show how many times a component appears as a child of one component.

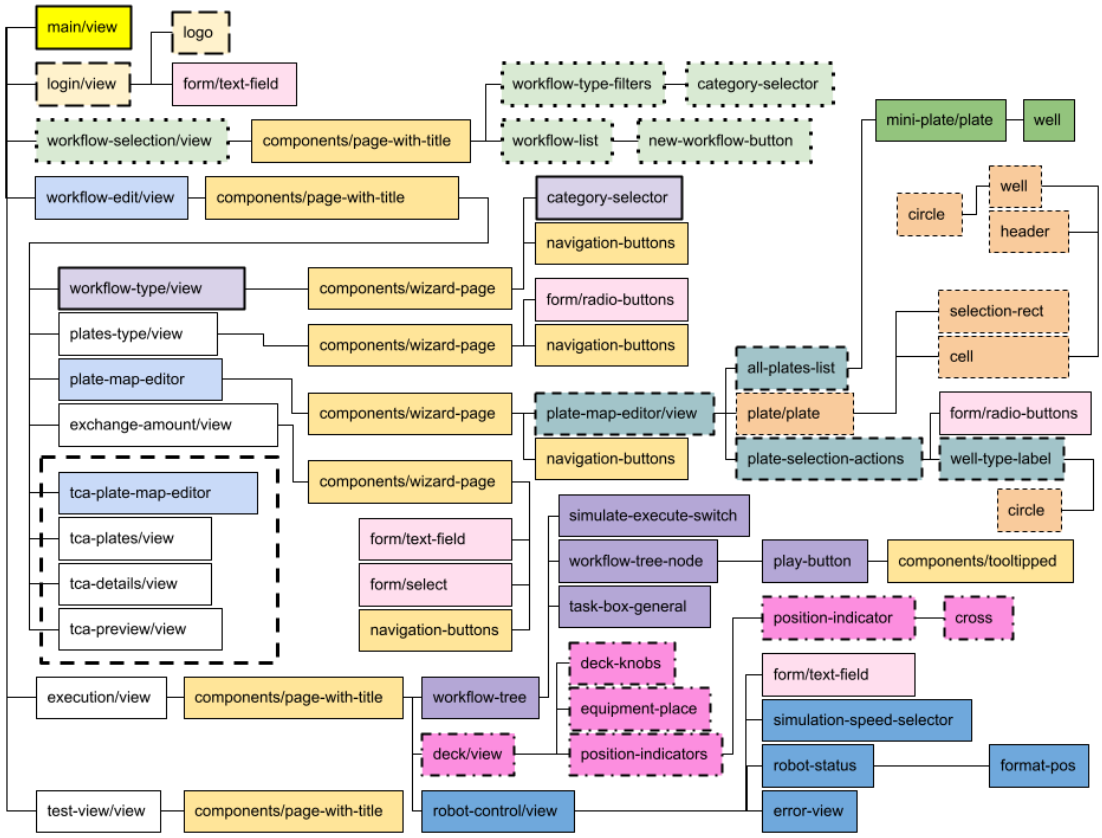

Figure 6: A diagram showing the component relations in the UI.

The TCA-wizard contains very similar components as the general workflow creation wizard, as can be seen from the diagram below (Figure 7).

The map editor is a GUI for planning plate maps for pipetting. The *tca-plate-map-editor* is almost the same component as the *plate-map-editor* in the previous diagram. The component is given different values for navigation

buttons, *next* and *previous*, but the *plate-map-editor/view* and its children are exactly the same as in the *plate-map-editor*. Currently the navigation is hardcoded into the components, but this is likely to change into a more flexible model in the future, possibly making this component redundant.

Plate information for sample and assay plates is gathered with form input elements in the *tca-plates* component. The data is updated into *app-db* whenever the inputs change. In future this data could be used to auto-generate plate maps for the plate map editor.

Details about equipment choice and workflow-specific parameters are gathered in *tca-details* component containing extra information in tooltip information modals along with the inputs.

On the preview page all the data gathered from the user is displayed on one page, *tca-preview*, so that the user can check if the parameters are correct before moving on to the deck planner. The deck planner, used to place the plates near the robot, is not part of the project scope.

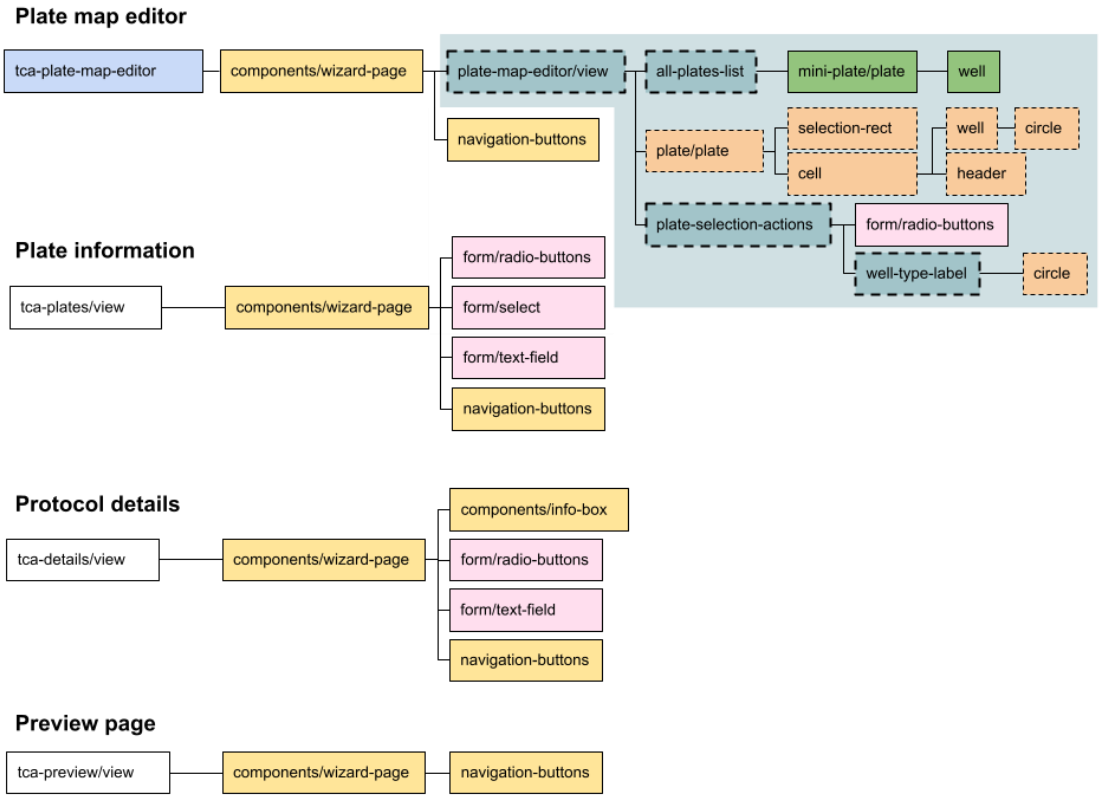

Figure 7: Components of the TCA-wizard.

4.3 UI design and prototyping

The original mock-up by the Customer Experience Designer was created using the Invision prototyping tool, but the following prototypes were created with Adobe XD. General views of designing and prototyping in Adobe XD are shown in the following images (Figure 8; Figure 9). Different views of the app are created as separate artboards that are connected in prototype tab to form a working clickable prototype.

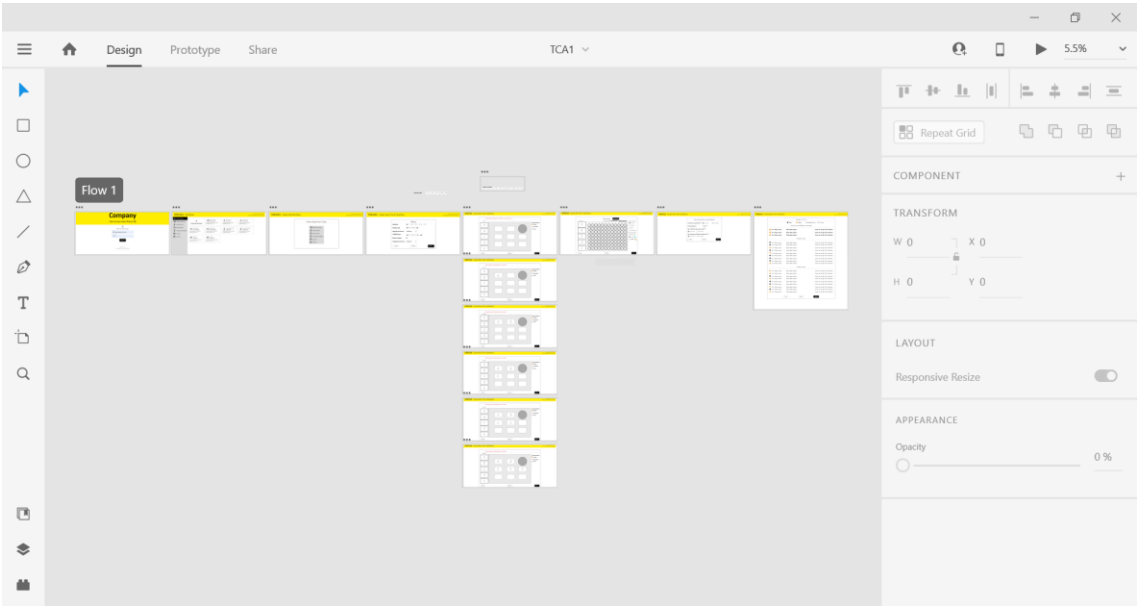

Figure 8: Design view in Adobe XD prototyping tool.

The following image (Figure 9) shows all the connections and effects between the prototype pages and components.

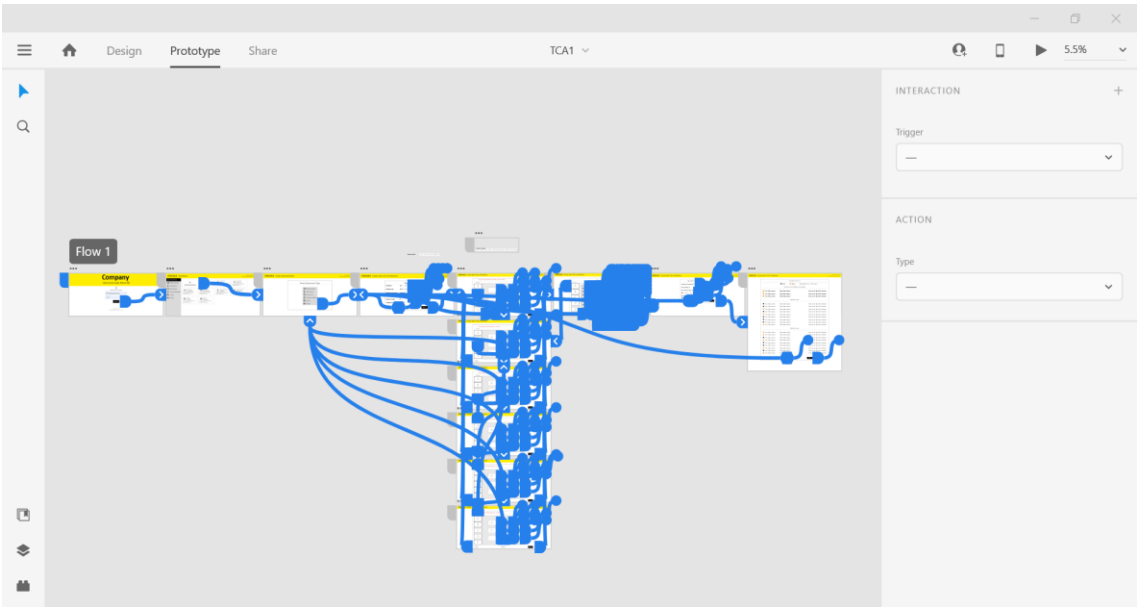

Figure 9: Prototyping view.

The minimalist visual guidelines of the company were unexpectedly updated in the middle of the project but are still fairly simple and straightforward. The

prototyping could be done with a very simple toolset of the Adobe XD software and the implementation did not require vast knowledge of style sheets.

As the UI in question is designed to be used for instructing a laboratory robot, most of the UI consists of different types of inputs the user provides, and the UI passes on. Finally, the parameters given by the user and pre-existing information about the workflow are compiled into a preview before sending the data onwards.

4.4 UI implementation with ClojureScript

Because the project concerned front end development with ClojureScript, most of the actions took place in the *project_gui/src/cljs/project_gui* directory. If that is not the case, it is mentioned in the text.

4.4.1 What needed to be done

Creating a new workflow wizard in the app consists of two larger sections that can be divided into smaller pieces. First creating and integrating the new workflow and second creating and integrating the content of the workflow.

Being a very modular functional framework, Re-frame can be used for building an app piece by piece without necessarily breaking the existing parts. This made it possible to actually advance in the order mentioned above. Before presenting the implementations, it would be wise to take a closer look at the working directories and pre-existing components of this project.

4.4.2 A closer look into the directory

The *main* namespace is the piece that determines which page view is rendered at any given time. It *subscribes* the information about the active view from app-db and returns the component to be rendered. For example, the following code snippet (Listing 9) shows how in the case of *active-panel* being *:edit-workflow*

the function returns a vector representing the component *workflow-edit/view*. In any other case it would return a vector representing an empty *div*.

```
(case active-panel
     :edit-workflow [workflow-edit/view]
     [:div])
```

Listing 9. Case function returning the appropriate page.

Workflow-edit is a view that is separated into a folder of its own, as opposed to the smaller and more general areas of the UI, for example the *login*, which has only one file of code. This folder has its own *core.cljs* which defines the contents of the view again by the information *subscribed* from *app-db*. Based on the result core should point directly to one of the page namespaces, for example, *tca-plates*. It is noteworthy that while the file names are written with underscore, Clojure namespaces are written with just a dash. This is not a typo and it matters in the code. Therefore, *tca-plates* refers to the namespace in *tca_plates.cljs*.

The *wizard-page* component and the navigation buttons for it are in *components*-namespace along with a few other regularly used components. Page components have to require it in their own namespaces if they are using it. The navigation buttons are Back-, Cancel- and Next -buttons at the bottom of the page. The dispatch functionality to update *app-db* is tied to the buttons, but the target page for each event has to be declared in each page component.

4.4.3 Adding a new workflow wizard

Creating a new workflow wizard to the project happens by following these steps:

1. Add the desired workflow in the *categories*-map in *workflow*-namespace in *cljc/project* folder.

   ```
   {:id :thesis, :label "Thesis-example", :icon :code, :description ""}
   ```

2. Create a new *.cljs* file in the *workflow_edit* folder and if the IDE does not create a namespace automatically, create a namespace by copying an existing one or creating an empty namespace according to the example. Remember that if a filename or path has an underscore, it is to be written as a dash in the namespace name.

```
(ns project-gui.workflow-edit.thesis)
```

3. Require Re-frame in the namespace:

```
(ns verto-gui.workflow-edit.thesis
     (:require [re-frame.core :as rf]))
```

4. Require the namespace just created in *workflow-edit.core* just like re-frame in the previous step.
5. Find the map called *step-info* and add the new namespace to the map by the name assigned in the require.

```
:thesis {:view [thesis/view]}
```

6. Also add it *to select-workflow-category*'s *case* function. The double colon refers to a keyword defined in the namespace.

```
(case category
     :tca-kit [::tca-plates/init]
     :thesis [::thesis/init]
     [::plates-type/init])
```

7. Return to the recently created namespace in *workflow_edit.* Require *components*-namespace from the parent folder and use it to create the view for the component. Required parameters, here replaced with “attributes” and “children”, can be found in the *components* namespace where *wizard-page* is defined.

```
(defn view [] [components/wizard-page {attributes} children])
```

8. Create *reg-event-db* to update the step in *app-db* when a user chooses the workflow and the view is initiated. The first keyword in *assoc* function refers to the destination of the data that comes after, in this case keyword *:thesis.*

```
(rf/reg-event-db ::init
     (fn [db _]
     (-> db (assoc :project-gui.workflow-edit.core/step :thesis))))
```

Creating more pages on the same workflow is done by following these instructions but skipping steps 1 and 6. Linking the pages to each other is done by defining the *:prev* and *:next* attributes of the wizard-page component in the namespace as in the snippet below. The attributes point to the namespace associated with the *init* function called when the button in question is pressed. If the keyword has value *nil* as in the snippet below (Listing 10), the button will not render at all.

```
{:title "Thesis"
     :prev nil
     :next [:project-gui.workflow-edit.thesis-two/init]}
```

Listing 10. Previous-button will not render with the value *nil*.

4.4.4 Creating the content

As mentioned before, Hiccup syntax is just Clojure vectors. As can be seen from the following snippets (Listing 11; Listing 12), this markup has the same elements as the usual JSX element.

```
<div className=”classy” style={{position: “absolute”, top: “20px”, left:
“45px”}}>
     “Hello World!”
</div>
```

Listing 11. Div element in JSX.

```
[:div.classy {:style {:position “absolute” :top “20px” :left “45px”}} “Hello
World!”]
```

Listing 12. Div element in Hiccup.

Considering this similarity it is fairly simple to create DOM content with basic knowledge of JSX or HTML.

4.4.5 Working with state

Within this project the updating of *app-db* with workflow wizard input data is done within the input components in *project_gui.form* and they are *subscribed* by whichever component needs them. The following code snippet (Listing 13) is from *form* namespace and it shows the event tied to the change in the radio buttons element where *re-frame.core* function *dispatch* gets triggered and dispatches *field-value-change* to store the values in *app-db* at the position the *data-path* points to.

```
:on-change #(rf/dispatch [::field-value-change data-path nil (-> % .-target .-
value)])
```

Listing 13. Dispatch happens when the value changes.

The function *field-value-change* is declared in the same namespace, hence the double colon, and uses *assoc-in* to execute the update of the *app-db* as shown in the following snippet (Listing 14).

```
(assoc-in db (concat [::data] data-path){:value value})
```

Listing 14. Function updating app-db.

This functionality, however, was integrated to the form element inputs. In this way a programmer rarely needs to work with dispatching data, they use the elements containing the effect. In the preview page, however, the data stored in *app-db* should be made visible for the user. The following snippet (Listing 15) shows a function that gets a data fraction called *:details* from *app-db*'s *form/data* section.

```
(rf/reg-sub ::details (fn [db _] (get-in db [::form/data :details])))
```

Listing 15. Getting data from app-db.

That function by itself does not yield the data needed on the page. The following snippet (Listing 16) shows how the data is *subscribed* and pieced into even smaller fractions: *:container-type* from *details* and from that *:value*.

```
(let [details @(rf/subscribe [::details])]
 [:div.preview-section
  [:div.data-line
   [:div "Containers for serial dilution"]
   [:div (:value (:container-type details))]]
  ])
```

Listing 16. Usage of a subscription to get specific piece of data.

4.4.6 Styling

Styling of the project was done technically with CSS, but using a library called Garden, which makes it possible to create CSS styles with Clojure vectors much like Hiccup.

The general style details are defined in *cljc/project_gui/style.cljc* by binding general values, like a map of company colours, to symbols. Symbols are a Clojure data type that reminds variable names in many other programming languages. In this way, they could be required in whichever namespace needed them.

The rest of the styling is located in *clj/project_gui/css.clj*. The styles are defined within a macro called screen as can be seen in the following snippet (Listing 17). The first element in the Clojure vector refers to an element in a manner known from CSS. Asterisks and hashes (referring to id) are strings, but tag names and classes are stated with keywords. Aside from the colons and the quotes, the selectors are precisely the same as in CSS. Use of the company colours required from a map in the *style* namespace can be seen in the last line of the snippet.

```
(defstyles screen
     ["*" {:box-sizing "border-box"}]
     ["#app" {:width "100%" :position "absolute" :bottom 0 :top 0}]
     [:body {:font-family "TTNormsPro-Regular,Open Sans,Helvetica,Arial,sans-
serif" :margin 0 :height "100%"}]
     [:.yellow {:background (:primary-yellow style/colors)}])
```

Listing 17. Heavily trimmed screen-macro.

The macro accepts an arbitrary number of parameters and generates styles out of them. Selectors can be used in a similar manner as in CSS. The following snippet (Listing 18) shows styling of two nested elements. The first is a *div* element with the class *info-wrapper*, containing a *div* with the class *info-button*. The style, however, is only applied when the cursor is hovered over the wrapper element. The second style is an *infobox div* element wrapping a simple classless *div* element.

```
[:div.info-wrapper:hover [:div.info-button {:background-color (:gray-40
style/colors)}]]
[:div.infobox [:div {:margin "10px"}]]
```

Listing 18. Styling of info-wrapper and infobox elements.

4.4.7 Running and deploying

Running the development build locally is done by opening two terminals and typing in the following commands one to each:

```
lein watch
lein garden auto
```

Deployment to production is done quite similarly with a few commands that build and upload the app to the Azure cloud platform. However, this is not part of the thesis project scope.

5 Results

5.1 User testing and feedback

User testing was done in the beginning of the project with an Adobe XD prototype to find out the general direction of the project. Unfortunately, due to the situation with secrecy and global pandemic, the testing could be done with only the Application Development Scientist.

The final user test for the thesis project was done on 17.3.2021 with the in-house Application Development Scientist. In addition to the TCA workflow wizard, the test included running workflows in simulation mode and on the actual robot. Therefore, the test documentation (appendices 1-3) has content clearly not related to the thesis project. The test was planned and analysed loosely following the agile development user testing principles introduced in a book called Kehitä kokeillen: organisaation käsikirja (39).

The user test documents (appendices 1-3) show test planning, test notes and analysis as separate pages as they are very distinct parts of the user test process.

The following storyboard (Figure 10) presents one of the user stories tested in this user test, excluding steps 5 and 6, which were tested with another workflow.

Figure 10: A storyboard depicting the ideal case of using the system.

5.2 Analysis

5.2.1 Test results

The test generally concentrated on the visual detailing of the UI but concerning the TCA-kit workflow part the focus was also on the usefulness of the content. It was discovered that the naming of the settings was quite unclear and clumsy for experienced lab personnel. The flow and the length of the workflow wizard, however, were considered appropriate. The tooltip-infoboxes containing detailed information about the related input field were considered very useful and, in fact, made it possible to finish creating the workflow despite the poor naming of the inputs.

Due to the modularity of a Re-frame project, updates and fixes to these issues and to most of the other issues mentioned in the test notes are simple to

implement. A few logical errors, like the sample count possibly being bigger than available wells, also should be fixed with fairly simple functions. The biggest visual problem, in fact, did not come up in the tests, but in the new brand style guides released in early March. The guides state that, for example, a brand application cannot have round or round edged elements. Style library Material UI used in the GUI rounds the edges of every element by default. These conflicts between the guides and the already existing application means the application style has to be revamped to be brand compliant in near future.

5.2.2 Testing

The test offered insight into the shortcomings of the GUI. However, it became clear that the product should be tested more with several test users not so closely in contact with the development. This test user had already seen multiple versions of the GUI before and seemingly found it difficult to pretend to be a mere tester during the user test and not suggest detailed improvements to the GUI during the test tasks. This and the familiarity to the logic and flow of the product might have diverted the attention from the general user experience. It would be useful to test the product more often with fresh users who have no previous experience of the product to acquire more accurate feedback on the usability and user experience. Testing with more users would also result in a greater variety of feedback and opinions. Testing with just one person over and over again will result in a very efficient and optimized product for that specific user but may overlook features and issues that would be important to other users. However, this type of testing is still better than to go on without testing entirely. Considering the current restrictions, it is very likely that tests with other test users would have to be organized remotely.

6 Conclusion

6.1 About moving from React to Re-frame

For people who mainly use LISP languages or are otherwise used to functional programming, Re-frame might be an easy way to take up React. In Re-frame, the React concepts are described using only Clojure data types and syntax and the style seeks to be as purely functional as possible - with Re-frame the illusion of purity is near perfection. For people who come from object oriented or imperative styles, getting a grasp of the philosophy of functional programming in Re-frame might require some effort. Functional programming, however, is a trending phenomenon: the newest React releases have had new features that support functional style programming and the immutable data libraries, like Immutable.js, have become popular. The perks of functional programming and immutable data are unquestionable when it comes to performance and they also reduce the possibility of human error in data handling.

The most noticeable, and therefore most intimidating part of adapting a new technology or language, is the syntax. Clojure syntax is very different from JavaScript, and that might make the new language seem alien and difficult. It is hard to convince a programmer that a certain language is clearer when they do not understand the syntax at all, which makes one of the Re-frame favouring arguments useless. Of course, this goes both ways: some Clojure programmers find HTML-like syntax of JSX in the middle of JavaScript too chaotic to consider React a comfortable option. This evidently is an issue of firstly the attitude and secondly the learning curve.

Popularity could be seen as the biggest single argument against adapting Re-frame instead of or with the usual JavaScript React. The size of communities surrounding these frameworks and their primary languages are barely comparable in size. The lack of support from a community may feel disheartening when learning something new, but it also should be mentioned that a smaller community is often less scattered and offers less bad advice than

a community of tens of thousands of members. Looking at the statistics of GitHub repos and StackOverflow entries it also seems that perhaps JavaScript has already passed the limit of useful, constructive community content and the vast number of possibilities becomes a nuisance instead.

6.2 About adopting Re-frame to project or company

A project like this GUI could very well be implemented with Re-frame if supervised by at least one person who has experience in the technology. Proceeding fast with the project would not be possible if everyone in the team has to first learn a new programming style, then a language and then the implementation. Knowing Clojure beforehand would of course flatten the learning curve. As an employer it should also be considered how few Clojure programmers there are to hire, compared to JavaScript programmers and people with experience with React. Scarcity of in-house Clojure programmers is a risk for any project implemented with a Clojure-based technology.

Learning and adapting the technology still has potential for paying off in the end through performance and eventual code clarity and modularity. When deciding upon the technologies of a project it should, however, be considered if there is time for the learning phase and if the risk can be afforded. Re-frame and especially ClojureScript have potential to become a very popular technology but currently they are still a very niche curiosity. ClojureScript would not be the first promising technology, which never really lifted off or died out quickly after a good start. On the other hand, the more businesses that join into pioneering the new technology the more popular it is bound to become. Perhaps the question to consider should not be if you should try developing with Re-frame or not, but rather, should you do it now.

References

1 ClojureScript. Online material. ClojureScript.org. <https://clojurescript.org/index>. Updated 11.3.2021. Viewed 15.3.2021.

2 Hickie, Rich. 2009. Clojure is Two! Online material. Clojure. <https://clojure.blogspot.com/2009/10/clojure-is-two.html>.16.10.2009. Viewed 22.2.2021.

3 Hickie, Rich. 2011. ClojureScript release. Online material. YouTube. <https://www.youtube.com/watch?v=tVooR-dF_Ag>. Viewed 10.2.2021.

4 Google Closure Library. Online material. ClojureScript.org. <https://clojurescript.org/reference/google-closure-library>. Updated 11.3.2021. Viewed 15.3.2021.

5 What is the Closure Library? Online material. Closure Library documentation. <https://developers.google.com/closure/library/>. Updated 24.5.2019. Viewed 13.2.2021.

6 What is the Closure Compiler? Online material. Closure Compiler documentation. <https://developers.google.com/closure/compiler>. Updated 20.8.2020. Viewed 13.2.2021.

7 Blackheath, Stephen & Jones, Anthony. 2016. Functional Reactive Programming. E-book. Manning Publications.

8 Brotherus, Robert. 2021. Tech Lead, Sartorius Biohit Liquid Handling Oy. Voice call 15.2.2021.

9 Brotherus, Robert. 2021. Tech Lead, Sartorius Biohit Liquid Handling Oy. Voice call 10.3.2021.

10 Using Hiccup to Describe HTML. Online material. CLJDOC. <https://cljdoc.org/d/reagent/reagent/1.0.0/doc/tutorials/using-hiccup-to-describe-html>. Viewed 15.2.2021.

11 Reagent: Minimalistic React for ClojureScript. Online material. Reagent-project documentation. <http://reagent-project.github.io/>. Viewed 20.1.2021.

12 Thompson, Michael. Re-frame. Online material. Re-frame documentation. <https://day8.github.io/re-frame/re-frame/>. Viewed 20.1.2021.

13 Thompson, Michael. A Data Loop. Online material. Re-frame documentation. <https://day8.github.io/re-frame/a-loop/>. Viewed 24.1.2021.

14 Thompson, Michael. Application State. Online material. Re-frame documentation. <https://day8.github.io/re-frame/application-state/>. Viewed 24.1.2021.

15 Thompson, Michael. Subscriptions. Online material. Re-frame documentation. <https://day8.github.io/re-frame/subscriptions/>. Viewed 10.3.2021.

16 Seibel, Peter & Margolin, Barry. 2005. Practical Common Lisp. E-book. Apress.

17 Converting a javascript function using Clojure. 2018. Online material. StackOverflow. <https://stackoverflow.com/questions/52010587/converting-a-javascript-function-using-clojure>. 26.8.2018. Viewed 16.3.

18 Olsen, Russ. 2018. Getting Clojure. E-book. Pragmatic Bookshelf.

19 Introducing JSX. Online material. React documentation. <https://reactjs.org/docs/introducing-jsx.html>. Viewed 10.3.2021.

20 Schæ, Jacek. 2020. Reagent with Juho Teperi. Online material. ClojureScript podcast. <https://soundcloud.com/user-959992602/s4-e2-reagent-with-juho-teperi>. 29.4.2020. Viewed 22.2.2021.

21 React Without JSX. Online material. React documentation. <https://reactjs.org/docs/react-without-jsx.html>. Viewed 10.3.2021.

22 Top IDE index. 2021. Online material. PYPL Index. <https://pypl.github.io/IDE.html>. Updated March 2021. Viewed 10.3.2021.

23 Best Integrated Development Environments (IDE). 2021. Online material. G2. <https://www.g2.com/categories/integrated-development-environments-ide?utf8=%E2%9C%93&order=popular>. Viewed 10.3.2021.

24 React Getting Started. Online material. W3schools. <https://www.w3schools.com/react/react_getstarted.asp>. Viewed 10.3.2021.

25 Setup Option 2: Local Development Environment. Online material. React documentation. <https://reactjs.org/tutorial/tutorial.html#setup-option-2-local-development-environment>. Viewed 7.3.

26 Paris, Dylan. 2015. Re-frame-template. Online material. GitHub. <https://github.com/day8/re-frame-template>. 2.3.2015. Updated 9.3.2021. Viewed 10.3.2021.

27 Redux Fundamentals, Part 2: Concepts and Data Flow. Online material. Redux documentation. <https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow>. Viewed 12.3.2021.

28 Vaughn, Brian. 2018. React v16.3.0: New lifecycles and context API. Online material. React blog. <https://reactjs.org/blog/2018/03/29/react-v-16-3.html>. 29.3.2018. Viewed 12.3.2021.

29 ClojureScript Libraries. Online material. ClojureScript.org. <https://clojurescript.org/community/libraries>. Viewed 14.3.2021.

30 1,000,000 million packages! now at 1,000,019 and growing - thank you npm community!. 2019. Online material. Npm. Twitter. <https://twitter.com/npmjs/status/1135968692062130176>. 4.6.2019. Viewed 16.3.2021.

31 Guerreiro, André. 2017. How comparing things is faster and simpler with immutability. Online material. No Solo Software blog. <http://nosolosoftware.com/comparing-things-with-immutability/>. 1.5.2017. Viewed 12.3.2021.

32 Hickey, Rich. 2015. Clojure, Made Simple. Online material. Oracle Developers. <https://www.youtube.com/watch?v=VSdnJDO-xdg>. 2.6.2015. Viewed 3.2.2021.

33 Hickey, Rich. 2020. A history of Clojure. In publication Wadler, Philip (ed.). Proceedings of the ACM on Programming Languages, June 2020, article No.: 71.

34 GitHub landing page. Online material. GitHub. <https://github.com/>. Viewed 4.3.2021.

35 Who we are. Online material. StackOverflow. <https://stackoverflow.com/company>. Viewed 4.3.2021

36 House, Cory. 2019. Why Your Team Should (Or Shouldn’t) Use React. Online material. Pluralsight. <https://www.youtube.com/watch?v=UNb44DKECIU&list=PLif6_xhXJh4ROWfVVwPAKr-3ll3_isuNf&index=3&t=0s >. 13.11.2019. Viewed 12.3.2021.

37 Introducing Hooks. Online material. React documentation. <https://reactjs.org/docs/hooks-intro.html>. Viewed 13.3.2021.

38 Human T Cell Activation Cell and Cytokine Profiling Kit. 2018. IntelliCyt Corporation. Document number 12660-A. Albuquerque: IntelliCyt Corporation.

39 Hassi, Lotta; Paju, Sami; Maila, Reetta. 2015. Kehitä kokeillen: organisaation käsikirja. E-kirja. Talentum Pro.

User test plan

User test March 2021 plan

What do we want to know	Why? (Hypothesis or user story)	Priority 1-3
Are the "Details" inputs understandable?	They might not be clear enough	1
Is "number of samples" a useful setting?	If we use autofill plates -feature, it might be?	1
What information the "Preview" page should contain?	So the user has a clear idea of what the robot is going to do	1
Is the flow of the wizard ok?	Usability -> Should we change the order of the pages?	1
Reformatting demo-workflow		
Is it easy to open "Xmas 24->96" workflow for execution?		
Is the workflow-tree understandable?		
Is the expanding and collapsing of the tree nodes intuitive?		
What is suitable detail level available in the tree?		
What attributes of a task should be displayed in its tree-node?		
Are the icons on the task-tree clear? Should icons be removed or changed?		
It is good idea to hide some task-node info when it is not selected?		
Is it easy to find how to execute a task?		
Does the deck view contain details that are unnecessary and/or distracting? (eg. equipment location number, row and column headers, deck pins)		

SHould some details be added to the deck view?		
Is the marking of pipetting head positions clear?		
Is the highlighting of plates / wells of selected node clear?		
Is the Simulation / Run on Robot texts and meaning clear?		
Is it easy to change speed of simulation?		
Is it easy to stop the simulation / run?		
Is it clear what "Reset Deck" does and when it is needed?		
Should the destination plate display current progress of pipetting (like it does now) or destination plate-map?		

User test notes and interview

User test March 2021 notes

Test information	
Testers	Söderholm,Sandra. Application Development Scientist
Date	17.3.2021
Place	Helsinki
Feedback	Interview
Recording	No

Tasks for the tester
Create new TCA workflow
Open Xmas 24-96 reformatting workflow
Execute the workflow in simulation mode
Stop simulation
Execute a single liquid-transfer in robot mode
Execute the workflow in robot mode

Interview questions	Notes
Did you succeed in the tasks?	Unsure about the single transfer lquid transfer
On scale 1-10 how difficult were the tasks?	Easy, 3
What would have helped you / made them easier?	Naming of the plates, execute/robot toggle clearer
Good things about the UI?	Hover for help buttons
What would you change?	Adding favourites and favourites on the front page.
Are we still missing something?	I don't think so
Points mentioned in the plan	**Notes**
Are the "Details" inputs understandable?	No. Short names for the steps and more grouping, although help bubbles help a lot. Add title “Preparation of standards”
Is "number of samples" a useful setting?	Possibly.

What information the "Preview" page should contain?	More subtitles and grouping
Is the flow of the wizard ok?	Yes
Reformatting demo-workflow	
Is it easy to open "Xmas 24->96" workflow for execution?	YES, I barely managed to tell the task before it was open...
Is the workflow-tree understandable?	Yes
Is the expanding and collapsing of the tree nodes intuitive?	Seemed to be, user went ahead and opened them instantly.
What is suitable detail level available in the tree?	
Is it easy to find how to execute a task?	Easy enough, choosing which one was harder.

Does the deck view contain details that are unnecessary and/or distracting? (eg. equipment location number, row and column headers, deck pins)	Deck pins (agreed on numbers too when asked about it)
SHould some details be added to the deck view?	
Is the marking of pipetting head positions clear?	
Is the highlighting of plates / wells of selected node clear?	
Is the Simulation / Run on Robot texts and meaning clear?	The toggle between them is a bit unclear, said the user, although had no problems recognizing them.
Is it easy to change speed of simulation?	It is easy but user found is confusing that the speed menu was not available when simulation was not running
Is it easy to stop the simulation / run?	It was easy to stop and run, but user was confused because there was no pause option: stop resets progress.
Is it clear what "Reset Deck" does and when it is needed?	No

Should the destination plate display current progress of pipetting (like it does now) or destination plate-map?	
	Categores confuse on the front page: user tries to first go through side bar but ends up in the same creation menu.
	Sample count can be bigger than the number of wells
	Filling of the plate map: user tries select-choose color method, which does not work
	Sample-id missing
	Mini maps should include the plate type (sample/assay)
	Label: Containers for serial dilutions → Standards serial dilutions.
	Label: Mix by pipetting → Mix by pipetting in Serial dilution
	Plates reset if user goes back to the plate form
	Execution mode details make screen blink too much
	Optionally hide low level steps
	Optionally hide stuff from deck view
	Pipetting on the robot: aspiration not done?
	Pipetting on the robot: dispensing too late

User test reflections

User test March 2021 reflections		
Acquired knowledge	**Follow-up plan**	**Priority 1-3**
Things should be named properly instead of "N" or "Y" if not testing help-button specifically.	Come up with better names and check them with Sandra.	3
Workflow visualization in a tree form seems intuitive to use.	Continue using this format, but clean up the lowest level tasks.	3
Extra information in the deck view did not bother an experienced user.	Make the deck view less noisy for a new user but keep the extra information optional.	2
Simulation controls are too scattered and/or unclear and not always visible (toggle, speed, stop, reset).	Group the controls and make them always visible.	1
Filtering on the left sidebar does not seem like filtering, it is assumed navigation.	Evaluate possibilities of a) tweaking the filter's appearance to be more clear b) turning the view into several "pages" and a navbar on the side.	2
BUG: sample count can be bigger than well count.	Fix the bug in logic OR remove sample count field (duplicate info)	2
Some users prefer selecting wells in plate map first and the contens after, some users prefer the other way around.	Evaluate and test the two options further.	3
Sample id and plate name missing.	Add the features.	3
BUG: plates suddenly resets if user goes back far enough in wizard mode.	Fix the bug in logic (save data or warn about losing the data).	2
Robot moves too hastily and skips or executes too late some of the steps: aspirating and dispensing.	Investigate the problem further to determine good solution.	1
Things good and beautiful:		

Wizard is short, just a few pages
Help tooltips
Simulation speed control (exists)