Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Independent Coursework

ClojureScript Application to Visualize Trends on the Platform Stackoverflow

Verfasser:	Doreen Sacker
Studiengang:	Internationale Medieninformatik (Master)
Matrikelnummer:	552936
Zuständiger Prof.:	Prof. Dr. Gefei Zhang
Datum:	06. Okt 2016

Contents

Abstract

This paper provides a basic foundation of the ClojureScript language, it's commands and the basic setup to start with. Further, an example application, written in ClojureScript is being introduced and explained in detail. The application visualizes questions from Stackoverflow to a given tag over a period of time. Due to the chronological sequence trends can be derived to the given tag, for example a language like ClojureScript.

1. Introduction

ClojureScript is the equivalent to Clojure for the web. Clojure was developed as a new Lisp dialect in 2007 and ClojureScript followed shortly after in 2011 [1]. Lisp is the oldest programming language, which still exists and is being used [2]. Lisp treats all code as data, this way it is possible to pass functions as variables [3]. Thus, everything in ClojureScript is an expression and everything has a value, so everything can be passed on to functions [1]. Because ClojureScript is a functional language, immutable and persistent data structures are core properties of the language [3]. ClojureScript compiles into JavaScript, this makes it possible to use ClojureScript in most environments smoothly, due to the large distribution of JavaScript. This paper provides a selection of ClojureScript features, however it does not cover all there is. The purpose of this paper is to provide a basic understanding of ClojureScript to encourage an easy start in ClojureScript.

2. ClojureScript Basics

This section explains the basics and core commands of ClojureScript. The website clojureScript.net is a good online environment to try out code sequences [1]. One of the best sources to find information about possible functions and variables is the ClojureScript Cheatsheet [2].

While learning ClojureScript it is important to keep in mind that it is a functional language and data structures can not be changed. Instead a new data structure is returned, if change is required. By applying structural sharing, where data can be shared between two version, just a minimum of data is copied and thus does not waste storage [2]. ClojureScript has to be pragmatic and break with functional programming when necessary [2]. For example to interact with a system, sometimes output is required. However, output is not pure functional, because side effects can happen. ClojureScript provides solutions for such scenarios, which will be explained in detail later [1]. ClojureScript is being kept as close as possible to Clojure itself. Therefore, the API and most sources related to Clojure apply to ClojureScript as well.

[1] http://clojurescript.net/
[2] http://cljs.info/cheatsheet

2.1. Syntax

There are two main characteristics of the ClojureScript syntax [2]:

1. Function calls in list brackets
2. Prefix notation of functions

Function calls in ClojureScript are written as lists in brackets( ). The first element in the list is the name of the function [2]. The parameters of the function follow after the first element [2]. The parameters are normally separated through whitespace, but comma, tab or new line are also possible [1]. There can be maximum as many parameters as the function takes. However, it is possible to pass less parameters to the function, this way a new function can be defined and applied to other parameters later. Comments can be written with two semicolons (;;) in front [4].

```
(+ 1 2 3)
;; -> 6
```

Code Example 2.1: ClojureScript Syntax [2]

2.2. Namespaces

The keyword *ns* defines the namespace of a file [5]. This is always the first command in a ClojureScript file and there can be only one namespace per file. The given name has to match the file name, the current package can be named with a dot in front [4]. The class cljs.core is loaded by default and references for additionally needed classes follow simplest after the name with the keywords *(:require ...)* or *(:use ...)* [4]. *require* loads libraries and *use* also refers to the library's namespace. The best way to import JavaScript libraries is the website CLJSJS [3]. The site provides JavaScript libraries for ClojureScript. All open source libraries for Clojure and ClojureScript can also be found on the website Clojars [4].

[3] http://cljsjs.github.io/
[4] https://clojars.org/

```
(ns coursework1.core
    (:require [coursework1.date :as date]))
```

Code Example 2.2: Namespaces [2]

For long names the keyword *:as* can be used to address the class in the current namespace with a shorter name. There is also the keyword *:refer*, if only a certain function or variable should be imported from a class. With the keyword *:exclude*, functions can be left out of the import [5].

2.3. Variables

Variables can be defined with the keyword *def* in ClojureScript. This creates a variable with a name, for example x. x can then be defined to represent the number 5. Because ClojureScript's data structures are immutable and persistent, x can not be changed to be another value later on, x always represents 5 from now on [4]. Further, a local variable in a function can be created with the keyword *let*. The keyword *def* is only being used top level in current the namespace.

```
(def x 5)
```

Code Example 2.3: Variables [2]

2.4. Functions

ClojureScript has first-class functions, which can be passed on as parameters to other functions and operators are functions as well [2]. ClojureScript allows to include many different characters into function names and thus increases readability. It is possible to include numbers, *, +, !, -, _, ', and ? into function names, but the first letter has to be a character [2]. An anonymous function can be defined with the keyword *fn* and can be called instantly with another pair of brackets around [2].

```
((fn [var1 var2]
     (+ var1 var2)))
```

Code Example 2.4: Anonymous Functions [2]

Another syntax option for an anonymous function is to use a hashtag. The percent symbol shows the position of the parameter, because no name for the parameter can be defined in this syntax. If there is more then one parameter, numbers are added to the percent symbol.

```
#(+ %1 %2)
```

Code Example 2.5: Anonymous Functions with # [2]

Naming a function is similar to naming a variable with the keywords *def* and *fn* for function. In the keyword *defn* are both keywords united, which makes it shorter and more readable [2]. Within other functions the combination of *let* and *fn* is used *letfn*. It is possible to add a description after the function name. With this docstring an automatic documentation can be generated by programs [2]. A ClojureScript function always returns the last element of the function.

```
(def add (fn [var1 var2]
           (+ var1 var2)))
```

```
(defn add "Docstring"
          [var1 var2]
          (+ var1 var2))
```

Code Example 2.6: Named Functions [2]

Functions can have multiple implementations with different numbers of variables in ClojureScript [2]. Thereby, the function does not have to be defined multiple times, the different implementations are just added as additional body of the function. With *&* a variable number of parameters can be passed for the same implementation [2].

```
(defn add "add variables or add one"
    ([var1] (+ var1 1))
    ([var1 var2] (+ var1 var2)))
```

```
(defn returnVar "returns last parameter"
      [& var]
       var)
```

Code Example 2.7: Named Functions [2]

2.5. Collections

Elements in collections are normally separated with whitespace, however tab or new line are also possible. In ClojureScript is a separation with comma possible, but it does not work with Clojure [2]. Collections can contain different types of values in one collection [2]. Additionally to some collections types a table with useful functions is provided in table 1.

2.5.1. Lists

Lists are used as function syntax in ClojureScript. However, they can be used as collections as well, if the lists are quoted they will not be evaluated. Another option would be to use the keyword *list* [2].

```
'(1 2 3 4 5)
;; -> (1 2 3 4 5)
```

```
(list 1 2 3 4 5)
;; -> (1 2 3 4 5)
```

Code Example 2.8: Lists [2]

2.5.2. Vectors

Vectors are written in square brackets in ClojureScript or with the keyword *vector*. One advantage of vectors is that their values can be reached easily with indices [2].

```
[1 2 3]
;; -> [1 2 3]
```

```
(vector 1 2 3)
;; -> [1 2 3]
```

Code Example 2.9: Lists [2]

A functionality of ClojureScript is the possibility of binding vectors on the left-hand side [2]. Parameters can be extracted and named this way conveniently. This makes it often possible to write short and clear functions.

```
(let [[fst _ thrd] [0 1 2]]
      [thrd fst])
;; -> [2 0]
```

Code Example 2.10: Sets [2]

2.5.3. Maps

Maps take additionally to the value a key, with whom the value can be accessed. Maps are written with curved brackets and the keys are written with a colon in front [2].

```
{:eins 1, :zwei "zwei"}
;; -> {:zwei "zwei", :eins 1}
```

Code Example 2.11: Maps [2]

2.5.4. Sets

Sets are written similar to Maps with curved brackets, but with a hashtag in front or with the keyword *set* [2]. Set is an unordered list and does not allow duplicates. An exception will be thrown if duplicates are inserted. Sets can be nested and contain different types [2].

```
#{:eins 1, :zwei "zwei"}
;; -> #{:eins 1 :zwei "zwei"}
```

Code Example 2.12: Sets [2]

2.5.5. Laziness

ClojureScript is designed to be very efficient, it only does as much as it has to [2]. For example if you say you want an infinite list and then print the first 100, ClojureScript will only generate the first 100 and not the whole infinite list.

Table 1: More Useful Functions for Collections [5]

(first coll)	Returns the first item of a collection
(rest coll)	Returns all items of a collection except the first one (a similar function is next)
(count coll)	Returns the number of items in a collection
(coll? coll)	Returns true if parameter is a collection
(empty? coll)	Returns true if the collections has no items (Careful empty returns an empty collection)
(conj coll x)	Returns a new collection with x added (Careful the position x is inserted can differ for the collection types)
(nth coll index)	Returns the value at the given index

2.6. Control Structures

ClojureScript offers the control structures if and cond for conditions and do for functions, which can have side effects.

2.6.1. If

The if function in ClojureScript contains three parameters. First the condition, second the expression, which will be evaluated if the condition is true and the third expression, which will be evaluated if the condition is false [2].

```
(defn valueOfX "if x smaller 0 return 0 else x"
    [x]
    (if (< x 0)
        0
        x))
```

Code Example 2.13: If [2]

2.6.2. Cond

Cond is used for more multiple conditions. There is also conp and case. Case is more efficient if the condition stays the same [2].

```
(cond (> x 0) "positive"
      (< x 0) "negative"
      :else "zero")
```

```
(condp = (keyword code)
        :es "Spanish"
        :en "English"
        "Unknown")
```

Code Example 2.14: Cond [2]

2.6.3. Do

Do is in ClojureScript similar to curved brackets in JavaScript. It is used if functions have side effects. Do takes an infinite number of variables, but returns only the result of the last expression [2].

```
(do
    (* 1 2) ;; this value will not be returned; it is thrown away
    (+ 1 2))
;; -> 3
```

Code Example 2.15: Do [2]

2.7. Recursion

Like in most functional programming languages loops are replaced through recursion in ClojureScript. If a function calls itself within it's definition, the function is called recursive [6]. The keyword *loop* defines a variable or a list of variables, under loop follows the implementation. With the keyword *recur* can be jumped back to the beginning of the loop. It is also possible to use just *recur* [2].

```
(loop [x 0]
    (println "Looping with " x)
    (if (> x 2)
        (println "Done looping!")
        (recur (inc x))))
```

Code Example 2.16: Recursion [2]

2.8. Higher Order Functions

A function, which accepts a function as parameter or returns a function is called a higher order function [2].

2.8.1. Map

The map function takes a function and a collection and applies the function to every element of the list. Then it returns a new collection with modified values [2].

2.8.2. Filter

Filter takes a collection and a condition and returns a new collection with all values from the initial collection, which suit the condition [2].

```
(filter odd? [1 2 3 4])
;; -> (1 3)
```

Code Example 2.17: Filter [2]

2.8.3. Reduce

Reduce takes a function a start value and a collection. The function is applied to the start value and the first element in the collection, then the function is applied to the result and the second element of the collection and so on. The given function has to take two parameters [2].

2.8.4. For

The for function returns sequences and takes a list and a expression. It can be combined with keyword conditions for the result. The keywords are *:let :while* and *:when* [2].

```
(for [x [1 2 3]]
      [ x(+ x x)])
;; -> ([1 2] [2 4] [3 6])
```

Code Example 2.18: For [2]

```
(for [x [1 2 3]
      y [4 5]
      :while (= y 4)]
  [x y])

;; => ([1 4] [2 4] [3 4])
```

Code Example 2.19: For [2]

2.9. Threading Macros

In ClojureScript Threading Macros allow to write readable code for nested functions. This way (f (g (h x))) can be transformed into (-> x (h) (g) (f)). There are different options on how to bind the result to the next function. With -> the result is given to the next function as the first argument, with -» as the last. The most flexible option here is as->, to insert the result on a specific parameter position in the next function [2].

2.10. Own Types

With the keyword *deftype* new types can be defined. The constructor call is made with the type name closed by a dot [2].

```
(deftype User [firstname lastname])
(def person (User. "Triss" "Merigold"))
```

Code Example 2.20: Own Types [2]

2.11. Protocols

Instead of interfaces ClojureScript uses protocols to provide type-based polymorphism. Protocols have a name and contain functions [2].

```
(defprotocol IProtocolName
    "A docstring describing the protocol."
    (sample-method [this] "A doc string associated with this function."))
```

Code Example 2.21: Protocols [2]

Further, it is possible to extend types with the keywords *extend-protocol* or *extend-type*. Hierarchies can be defined globally or locally between symbols and keywords or types [2]. The example shows that circle is defined as a child of shape.

```
(derive ::circle ::shape)
```

Code Example 2.22: Protocols [2]

2.12. ClojureScript vs JavaScript

ClojureScript is designed to fit JavaScript as good as possible. For example are ClojureScript Strings JavaScript Strings. Further, JavaScript objects and functions are easily included into ClojureScript. JavaScript data structures can be modified with the keyword *set!*, in contrast to ClojureScript data structures, which are immutable. JavaScript functions are not pure and can throw exceptions, these can be caught with a try/catch/finally block.

New JavaScript instances can be defined with the keyword *new* or with a dot after the JavaScript class.

```
(new js/RegExp "^foo$")
(js/RegExp. "^foo$")
```

Code Example 2.23: CS vs JS [2]

A JavaScript function can easily be called with a dot in front of the name of the function.

```
(def re (js/RegExp "^Clojure"))
(.test re "ClojureScript")
;; -> true
```

Code Example 2.24: CS vs JS [2]

To Access properties from JavaScript, a dot and a hyphen-minus has to be put in front of the name of the property.

```
(.-PI js/Math)
;; -> 3,14159
```

Code Example 2.25: CS vs JS [2]

JavaScript objects can be created with *js-obj* or with *#js*.

```
(js-obj "land" "de")
(#js {:land "de"})
```

Code Example 2.26: CS vs JS [2]

Converting JavaScript objects to ClojureScript objects and vice versa can be done with the commands *clj->js* and *js->clj* [2].

2.13. State Management

ClojureScript gives a way to manage the state of its data structures through atoms. These are objects, which are mutable structures. To get the current value of an atom object the keyword *deref* or @ is used [2]. The keyword *swap!* is used to change the data and the keyword *reset!* is used to fill the atom with a new object [2]. It is possible to watch the atoms and register to changes. Reagent uses this feature intensively for its *ratom*.

3. Project Setup

This section displays the easiest way to setup a ClojureScript project. However, it is also possible to setup a project manually without any additional tools required. The manual setup is attached in the appendix A for further reading.

3.1. Leiningen

Leiningen is a build tool for Clojure. It can be installed with most package managers or manually from the Leiningen homepage [5]. ClojureScript and Leiningen run on the Java Virtual Machine (JVM) and therefore need the Java Development Kit (JDK) installed to work [1]. ClojureScript needs at least Java 7 to work, Java 8 works as well [3].

Leiningen has several templates, which can be used to generate basic project setups [6]. The command for generating a project is *lein new [template] name* [7].

3.2. REPL

It is possible to build a ClojureScript project without REPL, however it is very convenient if you want to try out short snippets of code in the console. This way the file doesn't have to be reloaded or a huge function written to try something small. REPL stands for *Read* (input from the keyboard), *Evaluate the input*, *Print the result* and *Loop back for more input* [2].

3.3. Figwheel

The tool Figwheel builds ClojureScript code and reloads the browser automatically, when changes in the file are being saved [8]. This provides a comfortable environment to program with ClojureScript. Additionally, Figwheel reloads CSS and JavaScript as well [8]. It works with Leiningen, which has to be installed first. Another useful feature of Figwheel is that it gives feedback to the quality of the code and how to make adjustments. Figwheel brings a build in ClojureScript REPL, which gets noticed if changes in a file happen [8]. The easiest option is to generate a new project with a Figwheel Leiningen template with the command *lein new figwheel name* [8].

[5] http://leiningen.org/
[6] https://clojars.org/search?q=lein-template

3.4. Editors and IDEs

A good IDE for ClojureScript is Light Table. It highlights brackets in different colours, which makes it especially easy to find mistakes. Light Table offers instant evaluation of code and realtime feedback. Another IDE is IntelliJ with the Clojure plugin *Cursive*. ClojureScript can further be used with the editors Atom, Emacs, Sublime Text 2, or Vim.

4. Testing

ClojureScript testing can be included with *cljs.test*. It follows the functionality of the Clojure testing class [9].

```
(ns coursework1.core-test
    (:require [cljs.test :refer-macros [deftest testing is]]
```

Code Example 4.1: ClojureScript Syntax [9]

The keywords for testing methods are *deftest* and *is* [9]. The test can be run with the keyword *run-tests*. To see the test output on the console, the command *(enable-console-print!)* might have to be added up front [9].

```
(deftest test-do-something-x-y
(is (= (do-something-x-y 1 2) 3)))

(defn run-tests []
    (.clear js/console)
    (cljs.test/run-all-tests #"coursework1.*-test"))
(run-tests)
```

Code Example 4.2: ClojureScript Syntax [9]

5. ClojureScript Application

To check the suitability for daily use, a small application was programmed in ClojureScript. The objective of this application is to visualize Stackoverflow questions. These can be analysed and displayed for a certain timespan. Stackoverflow offers the possibility to ask questions on certain topics. For organisation the questions are tagged, for example

with the tag ClojureScript. The applications asks Stackoverflow for the quantity of questions, tagged with a certain tag, in a certain timespan. Because Stackoverflow is one of the largest platforms for programmers, it is assumed, that the quantity can display the trend of a tag in the programming community worldwide. The application visualizes these numbers in a chart. It is possible to enter a timespan and tags, which should be displayed. If more than one tag is entered, the numbers are interrelated. The core technologies used by the application are Reagent, libraries for asynchronous programming and communication and the charting library Chartist.

5.1. Reagent

Reagent is an interface between ClojureScript and React [10]. React is a JavaScript library developed by Facebook and Instagram [11]. It helps to efficiently change components in a website. React targets the view of an application and how to display content. React components can be used and individually rendered [11]. This way only the things that change have to be rendered again. Further, Reagent has the opportunity to define components just with ClojureScript syntax [10]. An example syntax for a simple button can be found below.

```
[:div
    [:p [:button {:on-click #(callfunction)} "Press!"] ]
]
```

Code Example 5.1: Reagent Component Syntax

Reagent has its own version of an atom to manage the state of data structures in ClojureScript. It is also possible to register to changes of atoms and if required, render corresponding components again. The components can be rendered with reacts render function. It takes a component as the first argument and a DOM node to display it in.

```
(defn run []
    (reagent/render [home-page](.getElementById js/document "app"))
)
```

Code Example 5.2: Reagent Render Syntax

5.2. Asynchronous Programming and Communication

The application sends a request message with the parameters timespan and tag to Stack Exchange. Two libraries are used to access the API. The *cljs-http* library for the request, which returns *core.async* channels and therefore needs the *core.async* library [12]. The go functions is a macro from the *core.async* library that examines the body of itself for any channel operations and will return its body then into a state machine. The library has operations in the go block for requests, which are put *>!* and take *<!* [13]. With *<!* it is possible to take the response from the channel. The request can be send with http methods, in the example application with get [12]. To access the Stack Exchange API and be able to send more than 300 request per day, this application was registered and got a key from Stack Exchange. The response is send as JSON [14]. The request is placed with a filter, that only the total number and no additional information is returned. This way only relevant data is send and can be processed further immediately.

```
(defn getFromUrl [kind from to tag]
    (go
        (let [response
            (<! (http/get (string/join
                           ["https://api.stackexchange.com/2.2/questions?"
                            "fromdate=" from
                            "&todate=" to
                            "&tagged=" tag
                            "&site=stackoverflow&filter=!bRyCgbjcxkJlK8"
                            "&key=sFL00JQUoK8d5n9GtHiGzg(("
                            ])
                {:with-credentials? false}))] (:total (:body response)))))
```

Code Example 5.3: Stack Exchange Request

5.3. Chartist

The total number of questions for a timespan for a certain tag is visualized with the chartist library. The selected timespan in the application is broken down into sections of 30 days and the total number is requested for these 30 days. A selected timespan of 60 days is broken into two requests and the results are being displayed in the chart through a line graph with two dots. The Chartist library is available for ClojureScript and can be added with Leiningen. With Figwheel and Reagent atoms, it is also possible to update the chart with new data, if a new request is made. An exemplary code snippet to draw a

chart with Chartist and ClojureScript can be found below.

```
(ns coursework1.chart
  (:require
    [reagent.core :as reagent]
    [cljsjs.chartist]
   )
    (defn show-chart [results months]
        (let [chart-data {:labels (mapv months)
                          :series [results]}
              options {:width  "700px"
                       :height "380px"}]
        (js/Chartist.Line ".ct-chart" (clj->js chart-data) (clj->js options))
        )
    )
```

Code Example 5.4: Chartist Syntax

6. Résumé

ClojureScript's list syntax makes code clear and structured. However, this also leads to many brackets in the syntax of a ClojureScript function and without an editor which highlights brackets, mistakes are hard to find. Furthermore, the community of ClojureScript programmers is still small. Thus, finding help for a problem can be difficult. One of the best networks for programmers is the Clojure team on the platform Slack. Many problems have not been solved yet for this language, which leaves a lot of room for programmers to try out new things and create new libraries. In my opinion, programming in ClojureScript is a great way to create web applications. However, you need some time to get used to the language and its environment.

A. Manual Setup

It is possible to set up a ClojureScript project without additional tools like Leiningen or Figwheel. If the JDK is already installed, the first step is to download the ClojureScript standalone JAR, which bundles the newest Clojure Version [7]. It supports simple scripting of the compiler and the bundled REPLs [3]. Afterwards, the projects structure has to be set up. A ClojureScript projects needs in its root directory a source folder and then the project folder, in which the core ClojureScript file is set up. The path for the ClojureScript file looks like this: src/test/core.cljs [3]. In this file a namespace and functions can be defined. To build the ClojureScript project a build file is required, the example code for the build.clj file can be found below. ClojureScript is just a library for Clojure, this is way Clojure code can be added easily [3]. For the Google Closure Library we definde *:main* as entry point for the application. Now the project can be build with *java -cp cljs.jar:src clojure.main build.clj* in the console under unix or under windows with *java -cp "cljs.jar;src" clojure.main build.clj* [3].

```
(require 'cljs.build.api)

(cljs.build.api/build "src"
                {:main 'hello-world.core
                 :output-to "out/main.js"})
```

Code Example A.1: Setup build.clj File [3]

To display a website an index.html file has to be added to the project.

```
<html>
    <body>
        <script type="text/javascript" src="out/main.js"></script>
    </body>
</html>
```

Code Example A.2: Setup index.html [3]

[7] https://github.com/clojure/clojurescript/releases

The Browser REPL is build into ClojureScript. Following a Repl file can be set up, a start code is below. Further, add *(defonce conn (repl/connect "http://localhost:9000/repl"))* to the core cljs file. With the command *rlwrap java -cp cljs.jar:src clojure.main repl.clj* the project is started under *http://localhost:9000* [3].

```
(require 'cljs.repl)
(require 'cljs.build.api)
(require 'cljs.repl.browser)

(cljs.build.api/build "src"
{:main 'hello-world.core
:output-to "out/main.js"
:verbose true})

(cljs.repl/repl (cljs.repl.browser/repl-env)
:watch "src"
:output-dir "out")
```

Code Example A.3: Setup build.clj File [3]

References

[1] S. S. VanderHart Luke: ClojureScript: Up and running. O'Reilly Media, Inc., 24, 2012. [Online]. Available: http://shop.oreilly.com/product/0636920025139.do (visited on 08/11/2016)

[2] A. Antukh and A. Gómez: ClojureScript unraveled. 25, 2016. [Online]. Available: http://funcool.github.io/clojurescript-unraveled/ (visited on 11/08/2016)

[3] R. Hickey: ClojureScript, 16, 2016. [Online]. Available: http://clojurescript.org/ (visited on 08/16/2016)

[4] K. Zachary: ClojureDocs, 16, 2016. [Online]. Available: http://clojuredocs.org/ (visited on 08/16/2016)

[5] C. Oakman: ClojureScript cheatsheet, 17, 2016. [Online]. Available: http://cljs.info/cheatsheet/ (visited on 03/10/2016)

[6] M. Lipovaca: Learn you a haskell for great good!: A beginner's guide. No Starch Press, 18, 2011. [Online]. Available: http://learnyouahaskell.com/recursion

[7] P. Hagelberg. (18, 2016). Leiningen, [Online]. Available: http://leiningen.org/ (visited on 08/18/2016)

[8] B. Hauman. (18, 2016). Bhauman/lein-figwheel, GitHub, [Online]. Available: https://github.com/bhauman/lein-figwheel (visited on 08/18/2016)

[9] R. Hickey. (21, 2016). Clojure/clojurescript, GitHub, [Online]. Available: https://github.com/clojure/clojurescript (visited on 08/21/2016)

[10] D. Holmsand. (19, 2016). Reagent: Minimalistic react for ClojureScript, Reagent: Minimalistic React for ClojureScript, [Online]. Available: https://reagent-project.github.io/ (visited on 08/19/2016)

[11] F. Inc. (19, 2016). React, React, [Online]. Available: https://facebook.github.io/react/docs/why-react.html (visited on 08/19/2016)

[12] r0man. (2016). Cljs-http, GitHub, [Online]. Available: https://github.com/r0man/cljs-http (visited on 08/19/2016)

[13] R. Hickey and contributors. (19, 2016). Clojure/core.async, GitHub, [Online]. Available: https://github.com/clojure/core.async/wiki/Getting-Started (visited on 08/19/2016)

[14] S. E. Inc. (2016). Stack exchange API, [Online]. Available: https://api.stackexchange.com/docs (visited on 08/19/2016)