Insert here your thesis’ task.
Bachelor’s thesis

Study of Utilising ClojureScript Technology in a Developer Company

Ondřej Altman

Supervisor: Ing. Robert Pergl, Ph.D.

11th May 2015
First, I would like to give special thanks to Ing. Robert Pergl, Ph.D. for his great advices and for supervising this thesis. Another special thanks go to my family and friends for their huge and endless support. Lastly, I cannot forget to thank my colleagues from the company Socialbakers a.s. who were giving me feedbacks for my thesis.
I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 11th May 2015
Czech Technical University in Prague
Faculty of Information Technology
© 2015 Ondřej Altman. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author’s permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Abstrakt

Tato práce se zaměřuje na shrnutí programovacího jazyka ClojureScript a jeho přínosu pro programování klientské části webových aplikací. Shrnutí je vytvořeno s ohledem na programovací jazyk JavaScript, který se na tento úkol používá nyní. Důležitou částí této práce je zhodnocení, zda firmy mohou profitovat z přechodu na ClojureScript z jazyka JavaScript.

Klíčová slova ClojureScript, JavaScript, Clojure, webové aplikace, funkční programování, Om

Abstract

This thesis focuses on review of ClojureScript programming language and its benefits for programming client-side web applications. Review is performed with respect to JavaScript programming language, which is currently used for this task. Important part of this thesis is an conclusion, whether companies could benefit from the transition from JavaScript to ClojureScript language.

Keywords ClojureScript, JavaScript, Clojure, web applications, functional programming, Om
# Contents

1. **Introduction** .......................... 1

1. **Goals and Approach** .................. 3
   1.1 **Goals** .................................. 3
   1.2 **Methodology and Structure** ........... 3

2. **Introduction of ClojureScript** ........ 5
   2.1 **Clojure** ............................... 5
   2.2 **ClojureScript** ......................... 6

3. **ClojureScript review** ................. 7
   3.1 **Language** .............................. 7
   3.2 **Tools** ................................. 14
   3.3 **Community** ............................ 17
   3.4 **Conclusion** ............................ 18

4. **UI programming with ClojureScript** ... 19
   4.1 **Problems with web UI programming** ... 19
   4.2 **React.js** .............................. 19
   4.3 **Om** .................................... 21
   4.4 **Other libraries** ........................ 24
   4.5 **Asynchronous events** .................. 25
   4.6 **JavaScript adoption** ................... 27
   4.7 **Conclusion** ............................. 28

5. **Application in ClojureScript** .......... 29
   5.1 **Application** ............................ 29
   5.2 **Implementation** ....................... 31
   5.3 **Testing** ................................ 33
   5.4 **Future Scope** .......................... 34
List of Figures

2.1 Usage of compilation targets for Clojure [1] . . . . . . . . . . . . . 6
3.1 Persistent vector . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.2 Editor popularity between ClojureScript programmers [2] . . . . . 17
4.1 Framework benchmark . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.1 Wireframe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.2 Screenshot of example application . . . . . . . . . . . . . . . . . . 34
6.1 Most loved programming languages by developers [3] . . . . . . . . 40
6.2 ClojureScript usage survey [2] . . . . . . . . . . . . . . . . . . . . 41
List of Tables

4.1 Framework comparison .......................... 24
Introduction

For creating client-side web applications, there is no escape from JavaScript. It is the only language supported in all of the popular web browsers. With fast and constantly improving JavaScript engines, web is very good platform for complex client-side applications. But JavaScript itself is far from being ideal language for this task. It is known fact, that JavaScript was designed in just 10 days. Instead of writing production code in JavaScript, growing amount of companies use one of the languages that are into JavaScript compiled. But most of those languages provide only minor language improvements and many pitfalls of JavaScript still remain unsolved.

ClojureScript is a promising modern alternative to JavaScript. It brings the opportunity of creating client-side application in functional programming paradigm and well-designed language. This thesis aims to provide overview of ClojureScript language and make conclusion, whether companies could benefit from transition to this language from JavaScript.
1.1 Goals

Main goal of this thesis is to perform a review of ClojureScript language and make a conclusion whether companies could benefit from the transition to this language from JavaScript. This goal is divided into four subgoals:

1. Perform review of ClojureScript technology with respect to JavaScript.
2. Summary the benefits and pitfalls of using ClojureScript and its libraries for creating client side user-interfaces.
3. Demonstrate how ClojureScript can be used for creating web applications.
4. Perform a management summary of transition from JavaScript to ClojureScript.

1.2 Methodology and Structure

In order to fulfill goals settled for this thesis, these tasks were set for specific goals:

**Goal 1** Provide review of ClojureScript language features, tools and community. (Chapter 2 and 3)

**Goal 2** Review ClojureScript libraries for creating web user interfaces and compare them with JavaScript ones. (Chapter 4)

**Goal 3** Demonstrate ClojureScript principles on example application (Chapter 5)
1. Goals and Approach

**Goal 4** Find a representative web development company, which uses JavaScript and perform study of transition from JavaScript to ClojureScript. (Chapter 6)
Introduction of ClojureScript

ClojureScript is a dialect of Clojure programming language. Before ClojureScript introduction, Clojure should be introduced.

2.1 Clojure

Programming language Clojure was created by Rich Hickey in 2007. Rich Hickey is a software developer with over 20 years of experience in various domains. Rich has worked on scheduling systems, broadcast automation, audio analysis and fingerprinting, database design in a variety of languages [6].

Clojure is designed to be a general purpose language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for Multi-threaded programming [7]. Another important aspect of Clojure is that it is a dialect of Lisp programming language. Main ideas behind Clojure are:

- It is a primarily functional language which embraces immutability
- It is a Lisp with all its benefits
- Emphasizes concurrency, to make creating multi-threaded applications more easy
- It is designed to be a hosted language

Clojure emphasizes programming with pure functions. Pure functions take values as parameters and produce return value, they are free of side effects. Clojure makes writing pure functions idiomatic with core data structures being immutable and persistent.
2. Introduction of ClojureScript

Do you use Clojure, ClojureScript, or ClojureCLR?

<table>
<thead>
<tr>
<th>Choices</th>
<th>Percentage</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clojure</td>
<td>68.06%</td>
<td>1313</td>
</tr>
<tr>
<td>ClojureScript</td>
<td>54.97%</td>
<td>736</td>
</tr>
<tr>
<td>ClojureCLR</td>
<td>2.94%</td>
<td>26</td>
</tr>
<tr>
<td>Total Entries</td>
<td></td>
<td>1339</td>
</tr>
</tbody>
</table>

Figure 2.1: Usage of compilation targets for Clojure [1]

From the first sight Clojure looks quite different to most popular programming languages today. Its Lisp syntax with prefix notation is still very unique in desktop and web programming. With the power of Lisp macros it is simple to create new language features.

Clojure is designed to be a hosted language. It can seamlessly integrate libraries from underlying platform. There are three main platforms that Clojure supports. Clojure can be compiled to run on JVM (Clojure) platform, CLR (ClojureCLR) or JavaScript (ClojureScript) environment (Figure 2.1).

2.2 ClojureScript

ClojureScript is a compiler for Clojure that targets JavaScript. It is designed to emit JavaScript code which is compatible with the advanced compilation mode of the Google Closure optimizing compiler [8]. It was created in 2011 and its main goal is to combine power of Clojure with reach of JavaScript. Today JavaScript can be run in various environments and its engines are fast enough to be a very promising platform. Also, there are environments, where JavaScript is the only option. JavaScript is the only programming language, that is standardized and supported in all modern web browsers.

There are languages like CoffeeScript, Dart, Typescript and many others, which are meant to be compiled to JavaScript and used in browser. These languages have better syntax than JavaScript and have features to make programming easier than it is in JavaScript. CoffeeScript is inspired by languages like Python or Ruby and has cleaner syntax with significant whitespaces, TypeScript adds type checking and some of the other utilities from functional languages. But none of those languages have significantly different semantics from JavaScript. ClojureScript is very different from JavaScript and has the semantics of Clojure language.
Chapter 3

ClojureScript review

3.1 Language

ClojureScript has many language features that are not yet present in JavaScript:

- Immutable data structures helps to create genuinely simpler systems than in JavaScript
- Its Lisp nature allows ClojureScript to have powerful system of macros.
- Sequence library provides set of functions which works on every collection data type
- Multimethods for polymorphism with own implementation of dispatch function
- Protocols for extending data types with new operations

In this chapter these features are described and explained. All of these features are common in Clojure and are not specific only to ClojureScript.

3.1.1 Persistent data structures

Most data structures in Clojure and ClojureScript are immutable. Immutable values are opposite to mutable variables. Once they are set, their value cannot be changed and only new value can be created, leaving the old one unaffected.

JavaScript has immutable strings and numbers, but the rest of the language is extremely mutable. Immutable values are very useful in context of parallel programming with threads. But they have advantages in single-threaded environments too.
3. ClojureScript review

3.1.1.1 Implementation of Persistent Data Structures

Figure 3.1 shows how persistent data structures are implemented in Clojure and ClojureScript. Persistent Vector (immutable version of JavaScript array) is represented as a trie with branching factor of 32 and depth of 7. In the Figure 3.1 is used branching factor of 4 for more compact visualization. Nodes of this trie contains 4 pointers to child nodes, and path to leaf can be represented as binary number, this number corresponds to an index in the vector. For example index 406 is represented in binary as 0110010110, which is path to the leaf and value in it. In order to make change in that index and leave old vector unaffected, all nodes in that path must be cloned. The leaf node is also cloned, but the value in the index is replaced by the new one. Conclusion of this approach is:

• Adding a new value (or any other modification) to a vector produces a new vector, leaving old one unaffected.

• Persistent data structures are constant time slower than mutable ones. For every ”mutation”, constant amount of nodes must be cloned.

• Most of the memory is shared between the old and new vector.

Other data structures in ClojureScript have the same properties of immutable values.
3.1.1.2 List

List is core data structure of all dialects of Lisp. Lists are evaluated to function calls or expanded with macros. To store data in list, it is needed to use escaped form to prevent evaluation.

```clojure
; first value in list is evaluated
(println 3 4)
; to prevent evaluation, list must be escaped with "'" symbol
(def value '(print "Hello"))
```

Listing 1: List

3.1.1.3 Vector

Vector is immutable version of JavaScript array. It provides fast random access and adding to end.

```clojure
(def numbers [1 2 3])
; new items are added at the end
(conj numbers 4)
; => [1 2 3 4]
```

Listing 2: Vector

3.1.1.4 Map

Map is a key-value data structure. It is analogy to JavaScript object, but map in ClojureScript can store even a complex keys. Because data structures in ClojureScript are hashable, they can be used as keys in map.

```clojure
{:a 1 :b 2} ; map with keywords as a keys
{:key "value"} ; string as a key
{1 "First" 2 "Second" 3 "Third"} ; number as a key
{[1 2] "First and Second"} ; vector as a key
```

Listing 3: Map

3.1.2 State

Sometimes mutable state is needed even in ClojureScript. Mutable state can be managed with Atoms.
3. ClojureScript review

```clojure
(def state (atom {:items [1 2 3]}))
; to access value inside, atom must be dereferenced
(println (deref state))
; => {:items [1 2 3]}
; @ is a shortcut for dereferencing
(println @state)
; => {:items [1 2 3]}
; swap! function mutates atom
(swap! state update-in [:items] #(conj % 4))
(println @state)
; => {:items [1 2 3 4]}
```

Listing 4: Mutable state in ClojureScript with Atoms

It is a convention in ClojureScript community to reflect mutation in name of the function. Every function that mutates Atom or does any other side-effect should have bang (!) at the end of its name.

3.1.3 EDN

EDN stands for an extensible data notation. A superset of EDN is used by Clojure and ClojureScript to represent programs and can be used by applications as a data transfer format [10]. EDN has many benefits over JSON 1 which is heavily used in JavaScript community. The biggest benefit over JSON is its extensibility of a new elements.

3.1.4 Sequence library

Every collection in Clojure is an implementation of Sequence protocol [11]. There is sequence library in Clojure core that has functions and macros, which works on every collection in Clojure or ClojureScript. This is great benefit opposed to JavaScript, where objects, arrays and strings have different functions. Clojure follows the principle that 100 functions on 1 data structure is better than 10 functions on 10 data structures [9]. Sequence library has functions and macros for:

- Creating a collection
- Filtering a collection
- Modifying a collection
- Adding to a collection

1 JSON stands for JavaScript Object Notation, it is a format for data objects transfer between server and web application.
Most of the collection algorithms can be implemented with Sequence library, recursion and loops are not used that often in ClojureScript as in other languages.

### 3.1.5 Namespaces

In large applications, namespaces are great way how to structure the application and prevent collision of function names. JavaScript doesn’t have built-in support for namespaces. Common practice how to emulate namespaces in JavaScript is with nested objects. Google Closure library uses this approach for namespaces and loading modules.

```javascript
goog.provide('example.utils');
goog.require('goog.dom');

example.utils.searchFor(id) {
    return goog.dom.getElementById(id)
}

example.utils.searchForClojure() {
    return example.utils.searchFor("clojure")
}
```

Listing 5: Namespacing in JavaScript with Closure Library

Nested objects are currently best option how we can namespace modules in JavaScript. But this approach is very verbose, because we have to write the whole path in nested object (Listing 5). ClojureScript follows the namespacing from Closure Library, but in addition we can make short aliases.

```clojure
(ns example.utils
 (:require [goog.dom:as d]))

(defn search-for [id]
  (d/getElement id))

(defn search-for-clojure []
  (search-for "clojure")
)
```

Listing 6: Namespaces in ClojureScript
The resulting code in ClojureScript (Listing 6) is shorter and more readable than it is in JavaScript.

3.1.6 Multimethods

Multimethods are one of the ways how to use polymorphism in Clojure and ClojureScript. Multimethods allows to pass own dispatch function (Listing 7). In JavaScript polymorphism is available only by dispatching on object type. That can sometimes lead to overuse of inheritance.

```clojure
(defn dispatcher [x]
  [(zero? (mod x 3))
   (zero? (mod x 5))])
(defmulti fizzbuzz dispatcher)
(defmethod fizzbuzz [true false] [x] "Fizz")
(defmethod fizzbuzz [false true] [x] "Buzz")
(defmethod fizzbuzz [true true] [x] "Fizzbuzz")
(defmethod fizzbuzz :default [x] x)
```

Listing 7: Possible implementation of FizzBuzz exercise in ClojureScript

Multimethods are very powerful concept of runtime polymorphism, but sometimes arbitrary dispatch isn’t needed and simpler approach can be used. Protocols are different way how to use polymorphism in Clojure and ClojureScript.

3.1.7 Protocols

Protocols are part of Clojure since version 1.2 and in ClojureScript they are from the very beginning. Protocols are set of function signatures, each with at least one parameter, that are given a collective name. It is possible to extend existing data types to implement a specified protocols with macro "extend-type". Extending protocols with new operation is also possible with macro "extend-protocol".
3.1. Language

ClojureScript can extend JavaScript objects with a new protocol.

```clojure
(extend-type js/RegExp
  IFn
  (-invoke
    ([this str]
      (re-matches this str))))
```

Listing 8: Extending JavaScript data type with a protocol.

Protocols are useful for extending JavaScript objects to implement Clojure functions for better interoperability (Listing 8). ClojureScript compiler uses protocols heavily for this purpose. Protocols are also way how to use polymorphism in Clojure and ClojureScript. The difference between protocol and multimethod polymorphism is in dispatch function. In protocol polymorphism only data type can be used to dispatch. This limitation leads to better performance over multimethods.

3.1.8 Macros

Macros are fundamental feature in all Lisp dialects. Because Clojure code is contained only from Clojure data structures, it can be treated as data. Macros allows to manipulate this data during compilation phase. Languages which don’t have support for macros cannot be extended and new language features can be added only in new version of language. In Clojure and other Lisp dialects everybody can create new language features with macros (Listing 9).

```clojure
(defmacro unless
  [pred & body]
  `(if (not ~pred)
     (do ~@body)
     nil))
```

Listing 9: Definition of a new macro

Good example of adding new language features with macros is a core.logic library, which brings support for logic programming to Clojure/ClojureScript [14]. This library shows how even whole programming paradigm can be added with macros (Listing 10).
3. ClojureScript review

Listing 10: Logic programming in ClojureScript with core.logic [14].

Many of Clojure and ClojureScript libraries uses macros for simpler usage and more readable code. Macros are very powerful, but they should not be overused. Because they modify code during compilation, they can make debugging harder. For this reason macros should be used wisely and functions should be used instead of macros every time it is possible.

3.1.9 JavaScript interoperability

Clojure and ClojureScript are designed to be a hosted languages. ClojureScript can use JavaScript libraries and communicate with JavaScript objects easily with built-in language support (Listing 11). ClojureScript can even extend JavaScript objects and libraries with protocols (Listing 8).

Listing 11: JavaScript interoperability

With great interoperability support, ClojureScript programmers can benefit from large number of JavaScript libraries and include them easily into their project.

3.2 Tools

For writing ClojureScript applications various tools are used. The most important of them is the ClojureScript compiler, which is essential for ClojureScript development. There many more tools that helps to make writing ClojureScript applications more pleasant and are covered in this section.
3.2 Tools

3.2.1 Compiler

ClojureScript is compiled to JavaScript in two steps. First compilation step is ClojureScript compiler. This compiler compiles Clojure code and emits JavaScript. In the second step this JavaScript code is optimized and minificated with Google Closure compiler.

3.2.2 ClojureScript compiler

ClojureScript compiler compiles Clojure code into JavaScript. The compiler is written in Clojure itself. First compile is slow (about 14s) because of JVM long startup, but for next changes in source code ClojureScript compiler is able to do very fast incremental compilation (about 100ms). Emitted JavaScript code is compatible with Google Closure compiler.

3.2.3 Google Closure compiler

The Closure compiler is a tool for optimizing JavaScript code\[15\]. It parses JavaScript code, analyzes it and performs several optimizations. Compiler can run in two modes: simple and advanced. Simple has short compile time and performs only basic optimizations.

In advanced optimization mode Closure compiler provides:

- Function and variable renaming
- Dead code elimination
- Global variable and function inlining

Dead code elimination and variable renaming helps to reduce size of JavaScript code. Function body inlining is also very useful for ClojureScript, because reducing amount of function calls has good impact on resulting performance.

3.2.4 Leiningen

Leiningen is project automation tool for Clojure and ClojureScript. It uses declarative configuration written in EDN (Listing 12). Leiningen can be used to manage projects dependencies, building projects and other tasks.
3. ClojureScript review

```clojure
(defproject leiningen.org "1.0.0"
  :description "Generate static HTML for http://leiningen.org"
  :dependencies [[enlive "1.0.1"]
                  [cheshire "4.0.0"]
                  [org.markdownj/markdownj "0.3.0-1.0.2b4"]]
  :main leiningen.web)
```

Listing 12: Example configuration in Leiningen [16]

3.2.5 REPL

REPL stands for "Read-Eval-Print Loop". It is a command line tool that
reads Clojure input from user, evaluates it, prints result and then waits for
another input (Listing 13). REPL can be used for exploring new ideas, testing
code fragments or remote connecting to running Clojure application.

Very useful is tool called Browser-REPL, it is a REPL which is connected
to browser and can evaluate ClojureScript. It compiles ClojureScript into
JavaScript and then runs it in browser. This is especially useful with libraries
like Om, which are covered in Chapter 4.

```clojure
cljs.repl> (+ 1 1)
2
cljs.repl> (map inc [2 3 4 5 6])
(3 4 5 6 7)
```

Listing 13: Example of REPL interaction

In JavaScript we can use browser console instead of REPL. Most of modern
browsers have very good web developer tooling for JavaScript. But on
server-side, there isn’t such support for interactivity. Node.js, which is the
most popular JavaScript server-side technology, has own version of REPL,
but it does not support module reloading which is essential for interactive
development.

3.2.6 Editor support

ClojureScript can be written in any text editor same as any other programming
language. Syntax highlighting is also available in most of the editors. Some
of the editors have special support for ClojureScript and are covered in this
section.

16
3.3. Community

3.2.6.1 Emacs, VIM

Both Emacs and VIM have plugins that integrate Clojure and ClojureScript REPL into editor. Emacs is very popular in Lisp community because Emacs itself is written in dialect of Lisp called Emacs Lisp.

3.2.6.2 LightTable

Another interesting editor is LightTable, which is written in ClojureScript itself. It is built on top of Node webkit and it has built-in support for Clojure REPL. LightTable has very interactive user interface and for that reason it is very good choice for learning Clojure or ClojureScript.

3.2.6.3 Cursive Clojure

Cursive is a plugin for IntelliJ IDE from JetBrains[17]. IntelliJ is a popular IDE for Java programming language. IntelliJ is known for its great syntax analysis. Cursive does the same powerful analysis on Clojure code, it provides smart code completion and refactoring capabilities. For that reason Cursive is very good option for maintaining large ClojureScript codebase.

3.3 Community

ClojureScript and JavaScript communities are very different. JavaScript is the most popular language according to repositories on Github[18], but there is a big diversity in JavaScript community. For most of the web programmers, JavaScript is one of their first languages they have to learn. For this reason, large part of community are beginners.

---

2 Github is an online project hosting service
3. ClojureScript review

ClojureScript community is different. Clojure and ClojureScript programmers are mostly experienced programmers with good knowledge of JavaScript. So despite much smaller community, a lot of interesting projects are created in ClojureScript. Many ideas are shared between JavaScript and ClojureScript community. When Facebook open-sourced their library called React.js for functional UI programming on web, ClojureScript community responded with own libraries built on top of React.js (library) and persistent data structures from ClojureScript \[19\]. After success of these libraries, JavaScript community created libraries for persistent data structures in JavaScript \[20\].

3.4 Conclusion

ClojureScript has many language features that makes it more pleasant language than JavaScript. With support of macros, new features to the language can be easily added. Community behind ClojureScript is already creating great libraries and new language features with macros. With Google Closure compiler integration, applications written in ClojureScript can be aggressively minified into reasonable size in production JavaScript. Obvious downside of ClojureScript is still much smaller community and fewer libraries than JavaScript offers.
UI programming with ClojureScript

ClojureScript is designed to be used for creating complex user interfaces. In this chapter is described how functional and declarative approach can be used for this kind of task. Declarative UI programming isn’t specific only to ClojureScript, but persistent data structures from ClojureScript goes extremely well with this programming paradigm and makes it even more powerful.

4.1 Problems with web UI programming

For rendering in client-side applications Document Object Model (DOM) is used. Problem with DOM is that it was designed for documents, not applications. For that reason, working with it is very inefficient. It is good practice to avoid manipulating DOM directly and use abstractions instead.

There are countless JavaScript libraries that handles DOM manipulation in some way. One of the most famous is jQuery [21], which is popular because it handles the differences between browser implementation of DOM functions. Problem with browser inconsistencies is still prevalent today, but situation is getting better than it was several years ago. With the rise of client side applications, low level libraries like jQuery are often replaced with complex UI frameworks. Most of them follows Model-view-controller (MVC) architecture and are based on object-oriented paradigm, but there are also libraries based on more functional approach.

4.2 React.js

React.js is JavaScript library for creating user interfaces. It embraces component design and declarative programming paradigm [22]. React was released
4. UI programming with ClojureScript

by Facebook in 2013 and reaction from JavaScript community was very controversial. React was different from most of the libraries that days. Unlike most other libraries it doesn’t use two way data binding between model and view in classic MVC architectures. React uses much simpler approach. React components are just function of the application state. For every change in application state, React re-renders whole view.

To make this process efficient, technique called Virtual DOM is used. During the component render phase React calculates Virtual DOM for each component and then performs diffing to real page DOM. After that only the minimal required updates to DOM are performed. This reduction of DOM updates has good impact on application performance.

```javascript
var CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am a CommentBox."
      )
    )
  }
});
```

Listing 14: React.js example component

React doesn’t use templates to represent HTML. Instead of that, elements are created with just JavaScript functions (Listing 14) that return Virtual DOM representation.

4.2.1 Performance optimization

Reducing amount of DOM updates by using Virtual DOM makes React.js fast[23]. But there are more optimizations possible. React components are just functions with state as input and virtual DOM as output value. By default components gets re-rendered even if their state remains the same. Decision that component should update itself is done in method shouldComponentUpdate (Listing 15). By default this method returns always true[24].

```javascript
shouldComponentUpdate: function(nextProps, nextState) {
  return true;
}
```

Listing 15: React.js default implementation of shouldComponentUpdate
We can provide own implementation of shouldComponentUpdate to describe when component should be re-rendered. For example we can update component only if the state of component has changed (Listing 16).

```javascript
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Listing 16: Component should be updated only when old state differs from new

With this implementation, component isn’t re-rendered when there is no change in its state. This optimization can speed up React.js greatly, but there is hidden problem in it. JavaScript objects and arrays can be mutated in place, so they cannot be easily compared just by reference. Only option how compare two large objects or arrays in JavaScript is to perform deep equal, which can be very slow.

### 4.2.2 Combining React with ClojureScript

Fast equality check is one of the big advantages of persistent data structures in ClojureScript. If we use React combined with ClojureScript, we will got many React optimizations for free just by using persistent data structures[25]. And because React components are just functions of application state, React can be easily combined with functional languages. ClojureScript is excellent match for React. There are many ClojureScript libraries that provides wrapper React and allows to use it in ClojureScript projects easily. Most popular is library called Om, which was created by David Nolen [19]. This library is described into detail in next section.

### 4.3 Om

Om is a ClojureScript library built on top of React.js [19]. Om uses React semantics for components and abstracts them with protocols (Listing 17). Every Om component is a record that implements these component life-cycle protocols. While Om keeps the semantics of React, it adds all the benefits from ClojureScript.
4. UI programming with ClojureScript

Listing 17: Simple Om component

(defn my-component
  "Simple component in Om"
  [message owner]
  (reify
    om/IRender
    (render [_]
      (dom/p #js {:className "msg"}
        message))))

4.3.1 Application state

Common source of errors in web applications is application state and its updates. Om simplifies handling with state by using global application state. This state is stored in single atom and every change in this atom forces re-render of the top level component. With this method, view is always synced with data in model (global state in atom) and there is no need for creating listeners (two way data binding).

Another benefit of using this approach is the possibility to easily serialize whole application state to some kind of storage. Application state can be stored into file and sent to another person over the internet (useful for testing). Same way state can be stored into browser local-storage, so if user refreshes the browser or even restarts the computer, the application persists the state from last session. With this approach programming some functionality that was very hard in JavaScript and OOP (Object-oriented programming) becomes very easy with Om, global application state and persistent data structures. For example we can save snapshots of global state into another atom and implementing undo/redo operations becomes trivial.

4.3.2 Performance

Important aspect of libraries and frameworks is their performance. When application scales, slow library may turn into performance bottleneck and lead into problems. Benchmark\[26\] of Om performance was done on project called TodoMVC. TodoMVC is a project where the same application of task manager is implemented in various JavaScript MVC libraries\[27\]. Om has own implementation of TodoMVC which was created by David Nolen\[28\].

Benchmark itself compares three popular JavaScript frameworks (Angular, Ember and Backbone) with JavaScript libraries with more functional approach (React.js, Mercury) and libraries written in different language - Om (Clojure-
4.3. Om

Benchmark consists of this scenario. User adds 100 todos, then marks them all as completed and then remove them one by one. Results of this benchmark (Figure 4.1) shows some very exciting facts:

- Om has better result than React. With intuitive thinking, Om should be slower than React, because uses slower data structures, and language that is compiled into JavaScript (not-native). Om is getting better result because immutable data structures allows performance optimalizations that were described in section 4.2.1.

- Om, Mercury and Elm have better results than other libraries. All of them uses functional paradigm and immutable data structures to achieve this result.

- Mercury (winner of the benchmark) is lightweight JavaScript library, which also adopts ideas from React.js and Om [29].

Another interesting metric is the total number of lines that was written for this application. In table 4.1 is more detailed comparison.
4. UI programming with ClojureScript

Table 4.1: More detailed comparison

<table>
<thead>
<tr>
<th>Library</th>
<th>Lines of code</th>
<th>Performance (ms)</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elm</td>
<td>160</td>
<td>293</td>
<td>Elm</td>
</tr>
<tr>
<td>Mercury</td>
<td>414</td>
<td>177</td>
<td>JavaScript</td>
</tr>
<tr>
<td>Om</td>
<td>297</td>
<td>371</td>
<td>ClojureScript</td>
</tr>
<tr>
<td>Backbone</td>
<td>420</td>
<td>495</td>
<td>JavaScript</td>
</tr>
<tr>
<td>React</td>
<td>421</td>
<td>694</td>
<td>JavaScript</td>
</tr>
<tr>
<td>Angular</td>
<td>197</td>
<td>1558</td>
<td>JavaScript</td>
</tr>
<tr>
<td>Ember</td>
<td>223</td>
<td>1809</td>
<td>JavaScript</td>
</tr>
</tbody>
</table>

4.4 Other libraries

Om isn’t the only ClojureScript library that uses React.js for creating user interfaces. Om is the most popular, but there are many more. This chapter covers some of them and describes their approach and its differences from Om.

4.4.1 Reagent

Reagent is another library built on top of React. Unlike Om, Reagent doesn’t provide semantics similar to React. Reagent simplifies component code just to render function (Listing 18). Another difference to Om is that Reagent emphasizes storing state in multiple atoms (Om uses single atom for whole application state). During the render phase, Reagent detects which atoms were dereferenced in component and then it makes a reactive link to those atoms. Every time atom changes corresponding components are re-rendered.

```clojure
(ns example
 (:require [reagent.core :as reagent :refer [atom]]))
(
def click-count (atom 0))
(defn counting-component []
 [:div
  "The atom " [:code "click-count"] " has value: "
 @click-count ". "
 [:input {:type "button" :value "Click me!"
   :on-click #(swap! click-count inc)]])]
```

Listing 18: Reagent example

In order to keep application perform well, it is needed to keep amount
of global atoms low. With high number of statefull atoms, the application would be hard to reason about same as applications written in imperative style with mutating objects. It is also important to prevent unnecessary atom dereferencing in child components. It is much better to dereference atom in top-level component and pass value to child. Otherwise a large number of reactive links would be created and application would be slower.

4.4.2 Quiescent

Quiescent is more low-level library than Om or Reagent (Listing 19). It doesn’t enforce style how to store application state. This is difference from the other two, because both Om and Reagent use atoms. Quiescent doesn’t either enforce way how virtual should be rendered. But there is strong opinion about internal component state, Quiescent doesn’t allow to components have own mutating state. This is more functional approach, when all state is global and render is just function of application state.

Listing 19: Quiescent example

```plaintext
(def Hello
  "A component that says hello"
  (component (fn [value]
    (d/div {:id "hello"}
      "Hello, 
      (d/span {:id "fname"} (:first-name value)) 
      " 
      (d/span {:id "lname"} (:last-name value)))
    {:name "HelloWidget"}))

Quiescent is less popular than Om or Reagent, but its pure functional approach makes it interesting option for creating web user interfaces.

4.5 Asynchronous events

User interfaces are full of asynchronous actions. Reasoning about and handling those actions is what makes programming user interfaces hard. Sadly, JavaScript doesn’t have built-in constructs for handling asynchronous actions. In ECMAScript 2015 there will be added Promise and Generators constructs, but it will took years until they will be supported in majority of browsers. Until then only option to handle asynchronous actions it through callbacks or JavaScript libraries that address this problem.
4. UI programming with ClojureScript

4.5.1 Callbacks

```javascript
var element = document.querySelector("#button");

element.addEventListener("click", function (event) {
    console.log("button was clicked!");
});
```

Listing 20: Asynchronous event in JavaScript, in this example callback is anonymous function that gets called after click on element

Using callbacks for handling asynchronous actions can lead to many problems. First problem with callback is that callback function is passed as argument to function call (input parameters are mixed with callback function. Listing 20). Even bigger problem occurs when one asynchronous action leads to another asynchronous action.

```javascript
var element = document.querySelector("#button");

element.addEventListener("click", function (event) {
    setTimeout( function () {
        console.log("button was clicked 1 second ago!");
    }, 1000);
});
```

Listing 21: Nested callback

Nested callbacks (Listing 21) are an anti-pattern in JavaScript. Many JavaScript beginners tend to write this style of code, which is known as "callback hell".

4.5.2 Core.async

Core.async is Clojure library that can be used in ClojureScript. It is a library that implements ideas from Communicating Sequential Processes (CSP) paper that was written by T. Hoare [32]. CSP use queue-like channels for handling asynchronous events and communication. There are two primitives in Core.async: channels and goroutines. Main operations on channels are putting and taking values from and into a channel. These operations are blocking, that means when value is putted into a channel, this operation blocks evaluation in the goroutine until same value is taken out of from that channel. With this concept, we can handle asynchronous events easily as if they were synchronous (Listing 22).
4.6. JavaScript adoption

Many of the ideas from Clojure and ClojureScript has been adopted by other languages. JavaScript is also influenced and there are many great libraries that brings some of the power of ClojureScript into JavaScript.

4.6.1 Immutable.js

Immutable.js is a port of Clojure persistent data structures to JavaScript (Listing 23) [20]. This library from Facebook is getting popular and used with React.js same way as ClojureScript datastructures are used in Om. It is good example how influential ClojureScript is.

```javascript
var map1 = Immutable.Map({a:1, b:2, c:3});
var map2 = map1.set('b', 2);
assert(map1 === map2); // no change
var map3 = map1.set('b', 50);
assert(map1 !== map3); // change
```

Listing 23: Persistent data structures with Immutable.js

4.6.2 js-csp

This library is still very experimental but it is getting more and more attention. It is a port of CSP from Go and Clojure languages. Implementation is based
of David Nolen’s (main contributor of ClojureScript language) blogpost about CSP [33].

```javascript
var c = csp.chan();

try {
    csp.go(function*() {
        console.log(yield take(c));
    });
}

try {
    csp.go(function*() {
        yield csp.timeout(5000);
        yield csp.put(c, "5 seconds elapsed");
    });
}
```

Listing 24: Core async example

Handling asynchronous actions can be much simpler with js-csp, but implementation relies on using generators (Listing 24), which is language feature that is still not supported in majority of browsers today.

4.7 Conclusion

In this chapter was described the way how ClojureScript handles creating rich user interfaces using functional language and persistent data structures. ClojureScript community adopted the JavaScript library called React.js and created libraries on top of it. Because React is written in declarative style of programming, it matches extremely well with ClojureScript and its data structures. Second big topic in this chapter were asynchronous actions in user interfaces. Core.async library for ClojureScript is elegant solution for this problem.

Persistent data structures in ClojureScript are big benefit in terms of creating user interfaces. Despite the fact that they are constant time slower than mutable data structures in JavaScript, they can be used with libraries like React.js with great benefits. The resulting performance of this combination is better than the usage of mutable data structures (Figure 4.1). Another big benefit of ClojureScript is Core.async library, which handles asynchronous actions in user interfaces.
One of the goals of this thesis is to demonstrate how web application can be created with ClojureScript. To fulfill this goal an example application was created. Although this application will be used in the future and deployed for real use, this thesis doesn’t focus on achieving a fully functioning application, but the aim is to demonstrate the basic principles in creating web applications in ClojureScript and its libraries.

5.1 Application

The application itself can be described as an interactive form, where user fill characteristics of specific building. After filling all of the needed parameters, data from the application are sent to server. Then the report is generated on the server side. The report contains information about how floods can damage the building and how some of the damages can be avoided.

5.1.1 Functional Requirements

User of the application should be able to:

1. Submit filled form to server
2. Save the current state of form and restore it later
3. View generated report

5.1.2 Non-functional Requirements

1. Easy extensibility and form structure change
2. Application should be supported in last 3 major versions of Google Chrome, Firefox, Opera and Internet Explorer
Figure 5.1: Wireframe
5.2 Implementation

With large amount of components (Figure 5.1) in the form and their interactions (switching between sections, input validation) this application with non-trivial user interface is good example to demonstrate ClojureScript and its libraries.

5.2.1 Used libraries

This section covers libraries that were used for implementation and explains, why they were chosen over the alternatives.

5.2.2 Google Closure Library

Google Closure Library is a large JavaScript library from Google developers. It offers wide range of functionality. Closure Library is designed to be compiled and minified with Closure Compiler, otherwise the library is too big for client-side applications.

5.2.2.1 Om

Om library was described in section 4.3. The library was chosen for its popularity and great documentation. Another benefit of the Om library for this task is that it follows the semantics of React.js library, so it’s more intuitive for the programmers, who have some experience with React.js (library) from JavaScript.

5.2.2.2 Cljs.test

Cljs.test is a library for unit testing in ClojureScript. With some configuration, Om components can be easily tested too. Cljs.test library is easy to integrate into project and tests are simple to write. In Listing 28 is example how a unit test might look like.

5.2.2.3 Chestnut

Chestnut is a Leiningen project template for client-side applications written in Om library [34]. It provides great development setup with essential tools already configured:

- ClojureScript compiler configuration for development/production.
- Development server with Clojure server-side libraries.
• Browser REPL and code reloading after changes.

5.2.3 Architecture

Main goal of the architecture of this application is to achieve great modularity and extensibility. Component structure of the application is described in single data structure (Listing 25), where data are tagged with information how this data should be represented. This data-driven approach allows easily manipulate the structure of the form and is commonly used in ClojureScript community.

Listing 25: Declarative component structure

```clojure
{:cid :section :text "Text" :name 4
 :children
  [{:cid :input/checkbox :value :unchecked :label "Label"}]}
```

This data structure is stored in server-side and then sent to the client, where it is transformated into Om components based on its component id. The transformation is done with single multimethod (Listing 26, `render-func`), which dispatches to appropriate implementation of `render-func` based on the component id (:cid).

```
; multimethod is used for polymorphic rendering of components
; based on their :cid keyword
(defmulti render-func
  (fn [component] (:cid component)))
```

Listing 26: Polymorphic render function

Implementation of single component is straightforward (Listing 27). Namespace with component consists of two important parts. First is component render function, which describe how component should be rendered. Second important part implementation of multimethod with keyword of the component.
5.3 Testing

Testing Om components requires writing unit tests in Cljs.test. Om components are functions of their input state, making them easy to unit test. We can mock input state, render the component into a created container, and then test the rendered HTML.

With this approach, the application is easily extensible without modification of existing code. Another benefit of this architecture is that it is easily testable. The next section explains more about this topic.

Listing 27: Implementation of a collapsible group form element in Om.

With this approach, the application is easily extensible without modification of existing code. Another benefit of this architecture is that it is easily testable. Next section explains more about this topic.

5.3 Testing

In order to test Om components in the application, unit tests in Cljs.test were written. Om components can be unit tested easily, because they are just functions of their input state. We can mock input state, render component into created container and then test rendered HTML.

Listing 28: Component testing with Cljs.test
5. Application in ClojureScript

We can easily scale this approach used in 28 to more than one component. With all state in global atom, even whole app can be rendered into container with given input and tested.

5.4 Future Scope

In next months more work will be done on this application. Goal of this thesis wasn’t to deliver complete application, but to demonstrate how ClojureScript application written in ClojureScript would be structured and tested. With component architecture and data-driven approach, which was chosen for this application, implementing rest of the application should be straightforward.

5.5 Conclusion

In this chapter was described how client-side application can be implemented in ClojureScript and its libraries. For the implementation was chosen Om library, which was described in Chapter 4. Many of the concepts of functional
5.5. Conclusion

programming are used in the application architecture. Application is still prototype with missing functionality, complete implementation belongs to the future scope of this thesis.
Management summary

In previous chapters were described techniques how ClojureScript can simplify process of creating and maintaining client-side applications. One of the goals of this thesis is to choose a representative company and formulate business criteria for transition from JavaScript to ClojureScript. The goal is covered in this chapter.

6.1 Company representative

Socialbakers is a Czech company with global reach. Mission of the company is to track public profiles on major social platforms to help marketers measure and optimize their social media performance\[35\]. With more than 50 developers and most of the codebase written in JavaScript, Socialbakers are an excellent candidate to represent company switching to ClojureScript.

6.2 Issues with JavaScript

Developers from Socialbakers were asked to write out a list of issues they have with JavaScript. Because JavaScript is used both in client-side and server-side in this company, answers are divided into two groups.

6.2.1 JavaScript in browser

- Behavior of \textit{this} keyword.
- Unclear rules for type conversion and comparison between different types.
- Numbers and NaN behavior ( IEEE 754 standard ).
- Lack of asynchronous constructs (will be solved with yield keyword in next version of JavaScript).
6. Management summary

- Missing support for immutable values.
- Missing support for interfaces or similar concept.
- Backward compatibility even in bad features of language. For example typeof null will return "object".
- Detecting if value is a number in JavaScript is very obscure.

6.2.2 Server side

- Exception propagation. When exception is raised in asynchronous code, it doesn’t propagate properly.
- Insufficient tools for debugging and profiling for server-side JavaScript.
- Missing support for static typing.
- Large number of third party modules are abandoned or low quality.

To eliminate some of the issues with JavaScript, several technologies are used in Socialbakers. In most projects CoffeeScript language is used for more compact syntax and some features that are not present in JavaScript, but CoffeeScript is just simple transpiler to JavaScript, it can eliminate common errors in JavaScript, but the language itself is not very different from JavaScript and still have same issues that JavaScript has.

Another technology that is used in many Socialbakers products are Google Closure Library and Compiler. Closure tools are part of ClojureScript, so for companies that are using it already, the switch is gonna be easier.

6.3 Demands on new technology

In order to make transition to new technology these business criteria were set:

1. Be able to develop faster and with less errors
2. Support for current JavaScript libraries
3. Easy to learn for developers
4. Great community around technology

Developers from Socialbakers are already considering to use ClojureScript on some side projects to explore its potential.
6.4 SWOT analysis of ClojureScript

This analysis summaries properties of ClojureScript technology.

6.4.1 Strengths

1. Well designed language
2. Immutable values for functional programming
3. Language can be easily extended with macros
4. Innovative ideas from community

6.4.2 Weaknesses

1. Compilation step
2. Smaller community, harder to find programmers

6.4.3 Threats

1. Popular languages will adopt all ideas from ClojureScript

6.4.4 Opportunities

1. Adoption from large companies
2. Multi-threaded programming in browser

ClojureScript has many advantages over JavaScript in language features, many of the problems that were described by a Socialbakers developers in Section 6.2 aren’t present in ClojureScript. Better language leads to less bugs in codebase, so for this reason ClojureScript should be considered over JavaScript. Figure 6.1 shows that Clojure is one of the most loved languages between developers according to survey done by StackOverflow company [3].

One of the downside of ClojureScript is that it adds compilation step to the build process of application (Weakness [1]). For small projects, this could be big overhead. On the other side, for big projects this isn’t problem at all, because these projects often needs Google Closure compiler for aggressive minification and optimalisation, so compilation would be part of build process anyway.
6. Management summary

Figure 6.1: Most loved programming languages by developers

<table>
<thead>
<tr>
<th>Most Loved</th>
<th>Most Dreaded</th>
<th>Most Wanted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swift</td>
<td></td>
<td>77.8%</td>
</tr>
<tr>
<td>C++11</td>
<td></td>
<td>75.6%</td>
</tr>
<tr>
<td>Rust</td>
<td></td>
<td>73.8%</td>
</tr>
<tr>
<td>Go</td>
<td></td>
<td>72.5%</td>
</tr>
<tr>
<td>Clojure</td>
<td></td>
<td>71.0%</td>
</tr>
<tr>
<td>Scala</td>
<td></td>
<td>70.6%</td>
</tr>
<tr>
<td>F#</td>
<td></td>
<td>70.1%</td>
</tr>
<tr>
<td>Haskell</td>
<td></td>
<td>69.5%</td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td>67.2%</td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td>66.8%</td>
</tr>
</tbody>
</table>

% of devs who are developing with the language or tech that have expressed interest in continuing to develop with it.

There are 60+ companies that are using ClojureScript for production use and almost half of the ClojureScript programmers use ClojureScript at work (Figure 6.1). In this section are described companies that are most known for transition to ClojureScript.

6.5 Companies using ClojureScript

6.5.1 CircleCI

Circle CI is continuous integration and continuous deployment service. They replaced Backbone framework and JavaScript and now are using ClojureScript with Om library. The whole client-side code of Circle CI is open sourced.

6.5.2 Prismatic

Prismatic is a social news discovery application. Both Clojure and ClojureScript are used in their product. With Om library on client-side they managed to make their code simpler, shorter, and easier to test and maintain.

40
6.6 Conclusion

It is hard decision for company to make switch to a new language. ClojureScript is not a good option for a project, where server-side dominates and where client-side implementation is just a few lines of JavaScript. ClojureScript is much more suited for large client-side applications with large codebase. Companies who are struggling with maintaining large JavaScript applications could benefit from transition to ClojureScript same as companies like Prismatic or CircleCI did.

Even if company doesn’t decide for a transition to ClojureScript, it can benefits from embracing ClojureScript as a language for learning functional programming. Functional programming in JavaScript is getting popular these days and with support of excellent libraries like React.js and Immutable.js, it can greatly simplify writing client-side applications. Workshops for employees with goal of embracing ideas from ClojureScript are a good way how to improve skills of developers and prepare them for upcoming functional libraries.
Main goal of this thesis was to perform a review of ClojureScript technology and make a conclusion whether companies could benefit from switching to ClojureScript form JavaScript. The goal was fulfilled with these steps:

Review of ClojureScript language, tools and community with respect to JavaScript was performed. (Chapters 2 and 3). This review summarizes basic concepts behind ClojureScript technology. Then were described techniques and libraries for creating web user interfaces in ClojureScript (Chapter 4). Special attention was paid to Om library, which is very popular in ClojureScript community. Results of performed benchmark (Figure 4.1) indicates, that libraries based on persistent data structures and functional approach have better performance than most of the mainstream JavaScript libraries. Techniques for creating client-side web applications in ClojureScript were demonstrated on example application. Application architecture and used libraries were described in Chapter 5. Full implementation will be delivered in next months and belongs to the future scope of this thesis. Last goal of this thesis was to create management summary of transition to ClojureScript. This summary was created based on cooperation with developers from Socialbakers company and can help other companies to decide whether they could benefit from switching to ClojureScript language (Chapter 6).

Functional languages are promising alternative for creating client side web applications and ClojureScript is definitely one of the most interesting. Further investigation in this area belongs to the future scope of this thesis.
Bibliography


[34] BRASSEUR, A. Chestnut. [online], [cit. 2015-03-02]. Available from: https://github.com/plexus/chestnut


[37] CircleCI. About Us. [online], [cit. 2015-04-28]. Available from: https://circleci.com/about

Appendix A

Acronyms

JVM  Java Virtual Machine
CLR  Common Language Runtime
EDN  Extensible Data Notation
JSON JavaScript Object Notation
REPL Read-eval-print loop
IDE  Integrated Development Environment
UI   User Interface
DOM  Document Object Model
MVC  Model-View-Controller architecture
HTML HyperText Markup Language
OOP  Object-oriented programming
CSP  Communicating sequential processes
AJAX Asynchronous JavaScript and XML
Appendix B

Contents of enclosed CD

- readme.txt .................... the file with CD contents description
- src................................ the directory of source codes
- app................................ implementation sources
- thesis.................. the directory of LaTeX source codes of the thesis
  - img........................... the directory of images used in the thesis
- text.............................. the thesis text directory
- BP_Altman_Ondrej_2015.pdf....... the thesis text in PDF format
- assignment.pdf ............ the thesis assignment in PDF format