Functional Reactive Programming on the Web

A Practical Evaluation

Christian Strand Young

Master of Science in Informatics
Submission date: December 2015
Supervisor: Svein-Olaf Hvasshovd, IDI
Co-supervisor: Kim-Joar Bekkelund, Bekk Consulting AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The web as an application platform is rising rapidly. With more complex solutions written in Javascript that run client-side, as well as server-side, challenges related to Javascript's asynchronous nature arise. This thesis explores and applies the Functional Reactive Programming paradigm (FRP) on the web as an alternative to traditional imperative programming. The potential of FRP in a context of the web is shown through case implementations of general practical real world problems that web developers may face.

Sammendrag

Webben som en applikasjonsplattform er i stadig vekst. Med flere komplekse løsninger skrevet i Javascript som kjører på både klientsiden, i tillegg til tjenersiden, følger flere utfordringer relatert til Javascripts asynkrone natur. Denne oppgaven utforsker og anvender paradigmet Functional Reactive Programming (FRP) på web-en som et alternativ til den tradisjonelle imperative måten å programmere på. Potensialet til FRP i kontest av webben er vist gjennom case-implementasjoner av generelle, praktiske problemer fra den virkelige verden som webutviklere kan støte på.

Preface

This document is the end product of the Master's Thesis of Christian Strand Young at the Norwegian University of Science and Technology (NTNU).

Acknowledgements

I would like to thank my supervisor, Svein-Olaf Hvasshovd for his help with proof reading and academic advice during my writing. I would like to thank Kim Joar Bekkelund for putting me on the path to an interesting and relevant subject. I would also like to thank Mikael Brevik for taking the time to discuss the subject and giving me advice on the code, and Espen Jacobsson for proof reading my thesis. Finally, I thank my dear girlfriend and family for their endless support during my time studying.

Contents

Abstract **i**

Preface **iii**

Acknowledgements **iii**

Contents **vi**

List of code examples **vii**

List of Figures **ix**

1 Introduction **1**

2 State of the Art **7**
2.1 The History of FRP 7
2.2 FRP on the Web Today 9

3 Motivation and Methods **11**
3.1 Motivation and Problems 11
3.2 Approach 12
3.3 Case Overview 12
3.4 Research Questions 14

4 Case Implementations **15**
4.1 Case 1: Simple Addition 15
4.2 Case 2: Logic Gates 21
4.3 Case 3: Real-Time Chat with WebSocket 25
4.4 Case: Complex Client Side Form Validation 33

5 Discussion **45**

6 Conclusion **49**

A Appendix 1: Simple Addition **55**

A.1 Simple Addition using Bacon and FRP 55

A.2 Simple Addition with jQuery 56

A Appendix 2: Logic Gates **58**

A Appendix 3: Real-Time Chat with WebSocket **61**

A.1 Server-side Node 61

A.2 Client-side Bacon 62

A.3 Client-side Jade Markup 63

A Appendix 4: Complex Client Side Form Validation **65**

A.1 Form Implementation with Bacon 65

A.2 PHP API Resource 70

List of code examples

1.1 Function with side effects . . . 3
1.2 Conventional imperative programming . . . 3
1.3 Reactive approach that cascades with change . . . 4
4.1 Three HTML input fields . . . 16
4.2 Imperative JavaScript addition . . . 17
4.3 Mapped and filtered Bacon property . . . 18
4.4 Combining and presenting data from two properties . . . 19
4.5 NAND gate implemented as a JavaScript function . . . 22
4.6 Declearing a Bacon property from a input field . . . 22
4.7 Mapping properties together to form new ones through a given function . . . 23
4.8 Assign the properties to respective input fields in the DOM . . . 24
4.9 Hello World with Express and Node . . . 26
4.10 Using a template engine with Express . . . 28
4.11 Listening and broadcasting through WebSockets . . . 29
4.12 Event stream that emits a message to socket . . . 31
4.13 Event stream that emits a message to socket . . . 31
4.14 One of the validation functions for the FRP form . . . 36
4.15 Creating the properties for the input fields . . . 36
4.16 Example of an AJAX request using HTTP GET . . . 38
4.17 Converting AJAX to a Bacon stream . . . 39
4.18 The chain of validating the username . . . 39
4.19 Validation of matching passwords . . . 41
4.20 Binding feedback to fields to the correct element . . . 42
4.21 Binding feedback on fields to the correct element . . . 43

List of Figures

1 Screenshot of the simple addition user interface 15
2 Screenshot of the logic gates user interface 21
3 Overview of the chat application's communication flow 27
4 Screenshot of the FRP web chat interface 30
5 Screenshot of the user interface of the form 35

1 Introduction

In recent years, the web as an application platform has grown rapidly. From simple home pages for personal and commercial purposes, the web has evolved into a rich interactive platform treasured for its availability and possibilities for the current developer. Web standards, such as HTML5 [27], CSS3 [20] and ECMAScript 5 and 6 [7], provide platforms that make it easy and accessible to develop modern applications, libraries and frameworks that form the web as we know it today.

The web has in later years taken a turn from the traditional way of a server responding with a block of markup and styles that is rendered by a client browser and displayed on their screen. Modern browsers implement ECMAScript standards [7] to what we know as JavaScript [11]. The JavaScript language can be used as a scripting language to allow dynamic features included in a web page, but only in recent years with the emerging standards has the real potential been revealed.

JavaScript runs on an engine in the browser, e.g. on the Chromium project's V8 engine created by Google [13]. The V8 engine compiles JavaScript to native machine code (such as x86-64) before executing it, in contrast to similar scripting languages like Python [35] that interprets bytecode.

The fast and modern JavaScript engines, in combination with client computers becoming more powerful, give us the possibility to move parts of the server application code to the client. This provides for a more seamless experience for the user, as manipulating the interface in the browser can be done directly without the immediate need for a new request-response cycle.

As the standards emerge, community-created libraries and frameworks for the web are growing. The largest package manager for JavaScript, npm (Node.js) [25], lists nearly 200 000 packages at the time of writing, while in comparison, the similar tool for Java, Maven Central, lists close to 122 000

[22].

JavaScript is a multi-paradigm language, and not statically typed and object-oriented like Java. The result is that libraries, frameworks and applications induce different coding styles and paradigms when writing code. This makes JavaScript easy for programmers of different backgrounds to quickly get started with, but it does not necessarily result in clean and efficient code.

JavaScript's nature, which includes asynchronicity and first class functions (the ability to pass functions as parameters to other functions), quickly introduces excessive nesting, non-pure functions (see Code example 1.1) and a mutable state of the application that the developer must handle manually. These practices can quickly get out of hand, and make the application unstructured and hard to maintain as complexity rises [10].

This thesis will focus on a paradigm called Functional Reactive Programming (FRP) that is meant to counter some of the aforementioned problems, in a context of the web. FRP is not a new concept and stems from a system developed by Conal Elliot and Paul Hudak in 1997 called FRAN, which was a collection of functions and data types for composing interactive animations [8]. Since then FRP has been used in, e.g., robotics, Graphical User Interfaces (GUI), music, and now web applications.

FRP bases itself on both functional and reactive programming. In functional programming, there are no side-effects, i.e., no mutations of state or observable interactions with other parts of the program. In a web application especially with user input, side-effects are somewhat necessary, e.g., when updating the Document Object Model (DOM), which is the tree structure of nodes that make up the elements of a website, as a result of triggering a change to an input field.

The following example, Code example 1.1, shows a function with a side effect instead of doing nothing else but returning a value, as it would be in a strictly pure function.

```
function plusWithSideEffects(x, y) {
    var element = document.getElementById('inputField');
    element.value = 'This is a side effect';
    return x + y;
}
```

Code example 1.1: Function with side effects

To preserve the ideas of functional programming in an environment such as the web, it combines these ideas with reactive programming. The idea of reactive programming is to have certain data types that contain dynamic values over time. Conventional imperative programming captures these dynamic values only indirectly, through state and mutations. Consider the following example: Code example 1.2.

```
var a = 1
var b = 3
var c = a + b
a = 2
console.log(c)
> 4
```

Code example 1.2: Conventional imperative programming

The moment "c" is declared, it holds the value of "a+b", but when the value of "a" is changed, the value of "c" does not cascade to the change. Such is the nature of imperative programming. This is because "c" is assigned the result when the expression evaluates, but holds no further reference to the

other variables. ”a” and ”b” can be changed later with no impact on ”c”, as shown in the previous example.

In comparison, reactive programming preserves the link between the referenced variables. Thus, when ”a” is updated, ”c” will follow up the change, as illustrated in Code example 1.3 below.

```
var a = 1
var b = 3
var c = a + b
a = 2
console.log(c)
> 5
```

Code example 1.3: Reactive approach that cascades with change

The idea in reactive programming is like that of a spreadsheet. For example, two data cells with integers (A1,A2), and another cell that holds a sum function:

```
A3:=SUM(A1-A2)
```

In a spreadsheet such as Microsoft Excel or Google Drive Spreadsheet, if either A1 or A2 is altered, we expect A3 to reflect this change.

Be it cells or variables, these are just instances of the same problem. The relationship between the cells, or variables, is not implicit or in the mind of the programmer. It is explicit, unbroken and forever. They are bound. These variables are referred to in FRP as observable behaviors. They are values over time. A behavior is the first essential element in FRP along with events. Events are a finite or infinite sequence of values over time. In various implementations, this is also known as an event stream.

When creating an FRP application, these streams are used to design the data flow of the application by, e.g., merging and transforming the values yielded by the streams. In combination with a strict functional paradigm, FRP thus allows declaring an immutable data-flow that does nothing other than what the code dictates.

This thesis aims to explore if we can apply the FRP paradigm to modern web programming in good faith, using existing implementations in JavaScript, and how well it can solve real world problems developers might face.

2 State of the Art

2.1 The History of FRP

The FRP paradigm introduced in the late 1990s by Hudak et al. [8], was initially referred to as Functional Reactive Animations (FRAN). FRAN targeted rich interactive animations, which at that time was a complex and tedious job to program. They believed the reason for this was the lack of high-level abstractions and the failure to distinguish between modeling and presentation. I.e. ”what” an animation is, and how it should present itself.

By letting the programmer express the ”what”, their idea was that the presentation should come partially automatically based on an explicit modeling approach. This is similar to the key point in reactive programming today, which concerns modeling data-flow as we touched by in the introduction.

Hudak et al. described the modeling process in four steps (see [8]):

1. **Define behaviors**

   The behaviors are the data containers or variables that change over time. When animating the radius of a circle, the radius value would be a good candidate to act as a behavior. As the animation progresses and the size of the circle changes, the radius value is changed over time, which corresponds to the definition of the behavior.

2. **Event modeling**

   Events, such as real world interactions, like mouse clicks or collisions in an animation.

3. **Declare reactivity**

   When events occur, we react to them and naturally change and update behaviors based on how we model our program.

4. **Polymorphic media**

   Different types of media have different properties to animate. For example image rotation, vectors in a 3D model, or sound mixing. These types of media are bound in a framework based on behaviors and events, but must be treated differently due to distinct properties each type of media has. This is similar to what we might expect in a web application today, as we have different elements with distinct properties. One must model the reaction to an event to the respective media or element.

The modeling process draws clear lines to how we interpret and create FRP applications today, but the concept has taken many turns in its time. FRAN and many other early adoptions like RealTime-FRP [40] and Arrowized FRP [24] was developed in the Haskell language [14], which caused problems. Arguably, the main concern is accessibility. In modern application development, Haskell is not very common. Drawbacks also include that of a technical nature. Some of the problems in FRAN and other adoptations are mainly related to space and time leaks, i.e., unexpected memory use that slows down computation. A reason for this is the lazy nature of Haskell. Haskell does not execute code before needed.

In recent years, there is one thriving Haskell FRP library, called called Reactive-banana [29], that is still maintained and used in applications. Reactive-banana has an efficient implementation that counters the problems from other types of FRP implementations, and it's website provides examples and documentation. However, such a library is not easily integrated with JavaScript

applications, as it requires separate compilation.

To conclude this brief history of the FRP paradigm: The ideas of the classic FRP are still preserved, and very relevant as we leap towards the modern platforms of the web.

2.2 FRP on the Web Today

The use of FRP on the web is fairly new. Thus, not much academic work or literature has been written. As with other up and coming technologies on web platforms, documentation and research are often to be found in articles, blogs and videos from everyday developers that create and explore in real life projects and current products.

One of the FRP-related projects that sprung out of academic work is the language Elm [5]. Elm is a standalone FRP language that compiles to HTML, CSS and JavaScript and was the result of Evan Czaplicki's Ph.D. thesis in 2012. Elm has grown to represent a large ecosystem with a package manager and community-created extensions and libraries. The language features what we now expect from a language labeled as an FRP library or language. For example, all data types that Elm uses are immutable. What we have defined as a behavior in this thesis is called a signal in Elm. Although the name is different, it represents the same: a value that changes over time. Elm uses the web as a platform, but will not be utilized in this thesis due to it being a complete language, and it is thus not suitable to our questions regarding integration of existing platforms and applications.

A project that is highly relevant for discussion is the Reactive Extensions (Rx) project by Microsoft [36]. Rx is a collection of libraries that support languages such as .NET/C#, C++, and JavaScript. The JavaScript library is named RxJS [30], and is, along with the other libraries in the Rx collection, open-source. This means that although Microsoft is the lead developer, any-

one can view the source code, and anyone can submit contribution requests.

RxJS is one of the most popular FRP libraries for JavaScript with nearly 15 million downloads in 2015 [32], and is used by well-known actors in the web application world, such as Netflix. Netflix is, by the time of writing, the world's leading Internet television network, with over 69 million members [23]. At Netflix, Rx is an essential building block to write complex asynchronous code and is part of their core systems [31].

RxJS could very well be the library used for research in this thesis, but the choice landed on an alternative called Bacon.js (Bacon) [4]. Despite the non-conventional, and non-descriptive name, Bacon is a fully-featured FRP library for JavaScript. Based on RxJS, it features the now familiar concepts of FRP: behaviors and events, although behaviors in Bacon are known as properties, and events are referred to as an event stream.

Even though RxJS is more popular and widely used in comparison with Bacon, which only has 240 000 downloads in 2015 [3], Bacon has documentation written in a more practical manner, which is an appealing factor for beginners in FRP.

3 Motivation and Methods

3.1 Motivation and Problems

The motivation for the research in this thesis builds upon the problems with conventional imperative programming, described in the introduction. Modern web applications developed in JavaScript face challenges when application complexity and scale increases, e.g., dynamic changes to the user interface triggered by events. Multiple sources for the events, e.g., mouse clicks, keyboard presses, and external AJAX-requests (see Chapter 4), contribute to complexity in the form of managing the state of the application, and keeping a declarative structure in the code.

In addition, the asynchronicity of JavaScript makes it even harder to achieve simplicity, as nested callbacks are quickly introduced to allow the asynchronous code to execute in the correct order.

The result is a low level of abstraction, meaning the developer must keep track of the flow and state of an application manually throughout the code.

FRP as a paradigm means to counter some of these problems with its reactivity. The reactive approach means that we can achieve a higher level of abstraction with a clear and precise design of the flow of the application. Problems we have discussed, such as nested callbacks and manual mutations of state can thus be handled more elegantly in theory. Combined with influences from the functional programming paradigm, we introduce functions such as ”map” and ”filter”, as well as concepts like pure functions. With its features, FRP might revolutionize how we do dynamic web development in the future. For now, it is at least worth exploring.

3.2 Approach

In this thesis, we aim to explore how the FRP paradigm applies to regular and everyday projects and problems that web developers in the industry face. Questions revolve around how simple it is to implement the theory into practice, the cost of introducing a new paradigm and how an eventual implementation scales, is maintained, and how it works in comparison with conventional methods.

To bring such questions to light, we have implemented several cases of well known and current parts of web applications. A somewhat similar research method was conducted by Khare et al. in their paper, "Geospatial event analytics leveraging reactive programming" [6]. They compared a reactive approach to an imperative approach in context of doing geospatial analytics.

In our early cases, we will illustrate differences with imperative solutions, and as this is part of the main motivation for the thesis, discuss our findings throughout with this in mind. Unfortunately, reactive and imperative implementations side by side on some of the more complex cases will be out of scope for this thesis.

3.3 Case Overview

Following is an overview of the case implementations. Details and illustrations will be found alongside the implementations later on.

- **Simple Addition**

  An introductory case to illustrate the difference between FRP and imperative coding styles. This case is a working implementation of Code example 1.1 from the introduction.

- **Logic Gates**

  Using FRP to program an illustration of logic gates. Logic gates are some of the primary building blocks of computer components like CPUs and RAM. In this case, we program gates such as AND, OR, and NAND, and couple them all together and provide a user interface to try out different values or signals into the circuit.

- **Real-Time Chat using WebSocket**

  In this case we make use of yet another modern feature of the web, namely WebSocket [37]. Using a WebSocket library for underlying communication, we apply FRP on top to manage the application and delivering messages to users.

- **Complex Client Side Form Validation**

  This is a common tedious task for a web developer. The case implements a complex instance of form validation done client side, and builds upon techniques from the previous cases as well as introducing many new. Several user inputs and internal dependencies, as well as external communication with other resources, makes this a full blown example of FRP in practice.

3.4 Research Questions

As we illustrate our cases through examples, illustrations, and implementation snippets, the following research questions are established to measure each case:

1. **Code quality**

   Although somewhat subjective, the quality of the code is essential. Is the overall readability and size of the code without serious deviations? Are we able to apply the nature of FRP that solves many of our discussed problems in theory without falling back to usual habits?

2. **Maintainability**

   How easy is it to maintain the code? Is it trivial for another developer to enhance or expand the solution with new features or properties?

3. **Limitations**

   Being limited to the new paradigm, can we achieve our demands for the application implementation wise according to our specifications?

4 Case Implementations

The following sections are implementations of the cases from the previous chapter in the same order. Each implementation section is structured as follows: An introduction with an illustration/figure of the problem or specification of what is to be implemented, the implementation with code snippets to illustrate the different techniques, and finally a small discussion with regards to our research questions defined in Chapter 3. A discussion of the results that follow, can be found in Chapter 5.

All cases are implemented in HTML5, CSS2/3, and JavaScript running in Google Chrome [12] on MAC OSX Yosemite [2]. The jQuery library [18] is used alongside native JavaScript to simplify some of the code. The FRP-related code is provided by the Bacon.js library [4], which is used exclusively. Other technologies, libraries and third party software used is specified in each case.

4.1 Case 1: Simple Addition

This case illustrates a simple dynamic interface that changes based on what the user provides as input. The interface consists of three regular HTML input fields, where the values of the first two are summed, and the result shown in the third. The user interface is illustrated in Figure 1 below.

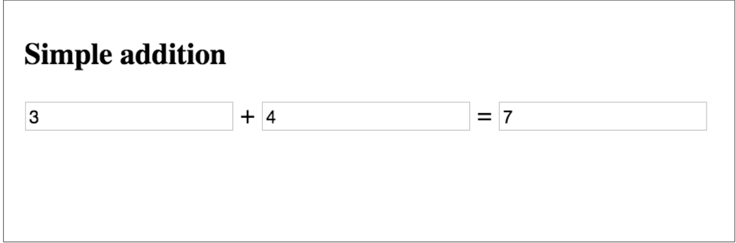

Figure 1: Screenshot of the simple addition user interface

When either of the first two values changes, the result cascades to the change. Traditionally, we would add an event listener to each of input fields that are eligible for change. When a change occurs, we would calculate the sum and update the third input field with the value.

Note that parts of the skeleton that are needed for the code snippets to run is omitted (consult Appendix 1 for runnable code regarding this case).

In addition to the JavaScript, the HTML elements that the JavaScript manipulates must also be defined. In later cases, such HTML-code might be omitted. Consult the full examples in the appendices for better understanding. For this case, we will show the elements in Code example 4.1 as this is an introductuary case and the amount of code is small.

```
<input type="text" id="a" value="0"> +
<input type="text" id="b" value="0"> =
<input type="text" id="c" value="0">
```

Code example 4.1: Three HTML input fields

These are regular HTML input fields that allows the user to change their value. Now, consider the JavaScript that makes them dynamic according to the specification, solved imperatively in Code example 4.2 following.

```
1  $('#a').on('keyup', function () {
2      var a = parseInt($('#a').val());
3      var b = parseInt($('#b').val());
4      $('#c').val(a + b);
5  });
6
7  $('#b').on('keyup', function () {
8      var a = parseInt($('#a').val());
9      var b = parseInt($('#b').val());
10     $('#c').val(a + b);
11 });
```

Code example 4.2: Imperative JavaScript addition

First, we declare two event listeners. The event listener works as follows:

```
when "some event" triggers from "element" => call function
```

The first listener on line 4 listens to "keyup" events, i.e., key strokes, from input element "a". If an event occurs, an anonymous function (i.e., a lambda function or a nameless function, [39]) is called. Such a function in this context is also referred to as a *callback*. The code inside the callback is then executed, which reads the value from "a" and "b", sums them and sets the value of the last input field to the result.

There are not many concurrency problems here as there is only one level of nesting. One might notice that the code around the callbacks could be rewritten to avoid the redundancy, but that is not important at this point.

Now, let us see how this could be done using FRP and Bacon. Consider the following Code example, 4.3, which illustrates how one of our input fields can be programmed.

```
var a = $('#a')
    .asEventStream('keyup')
    .map('.currentTarget.value')
    .map(parseInt)
    .filter(isNumber)
    .toProperty(0);

var b = $('#b')
    .asEventStream('keyup')
    .map('.currentTarget.value')
    .map(parseInt)
    .filter(isNumber)
    .toProperty(0);
```

Code example 4.3: Mapped and filtered Bacon property

We declare an event stream for each of the input fields. The streams will yield the changes made. This looks familiar to what we did in the imperative solution. The difference is that the ”asEventStream” function comes from Bacon and renders ”a” and ”b” as event streams that yields values over time. With the ”map” function, we are saying that every time there occurs an event on the stream, the value that stream should yield is the value of the input field. The goal is to combine the values from the two streams and from there form the result value.

In the imperative solution, data from the input fields were run through the ”parseInt” function as the values on the input fields are strings. They must be converted to integers to pass through our sum function. With Bacon, we can achieve the same function by adding another map to the chain as shown on line 4.

Note that this function could return NotaNumber(NaN) that will result in wrong computation. A way to mitigate this is to introduce a new function ”isNumber”. We can then chain this on, by using the ”filter” function, as done on line 5. This way, the stream will only yield a result if the value was a number and passes through our new filter.

For the event stream to hold onto its current value, as we have no subscribers to this stream, we must convert it to a property, or behavior as it was called in the classic FRP. In Code example 4.3, this is done at the end on each stream. The streams, now properties, will hold the value 0 (as an initial value passed as an argument to the function) until the value in the input field changes.

The complete properties can now be combined into a new property and presented to the user in the last input field shown in Figure 1. The following example, Code example 4.4, shows an example of how this can be done.

```
1 var c = a.combine(b, sum);
2 c.assign($('#c'), 'val');
```

Code example 4.4: Combining and presenting data from two properties

Our result property ”c” is now ”a” and ”b” combined *through* the function ”sum” (function that sums two numbers) as shown on line 1. Due to the reactive nature of Bacon, if either ”a” or ”b” changes, ”c” will hold the updated sum.

We then assign the property to an element to display the result to the user. Every time ”c” changes, its value is assigned to the 'val' property of the input field ”c”. The flow of data is defined from top to bottom: from the user input, through transformation of the data, and in the end displayed to the user.

As with the imperative example, the FRP apporach may also seem redundant code wise and could also be refactored. An example of this is shown in Case 4 where stream declarations use a help function.

4.2 Case 2: Logic Gates

The task of presenting highly stateful information to a user can be tedious to program. In this case implementation, we illustrate this by creating a circuit of logical gates. As the circuit's values change, the result is presented to the user in real-time. The circuit has two inputs and consists of the gates AND, NAND and OR. The coupling between the gates adds another level of complexity, as the result of one gate may be the input of another.

To simplify, we display the state of inputs, intermediate results and the end result in HTML input fields under the drawing. If this were to be implemented as a product, e.g., for a customer, one might take the time to make it so the values are displayed directly on the circuit for a better understanding.

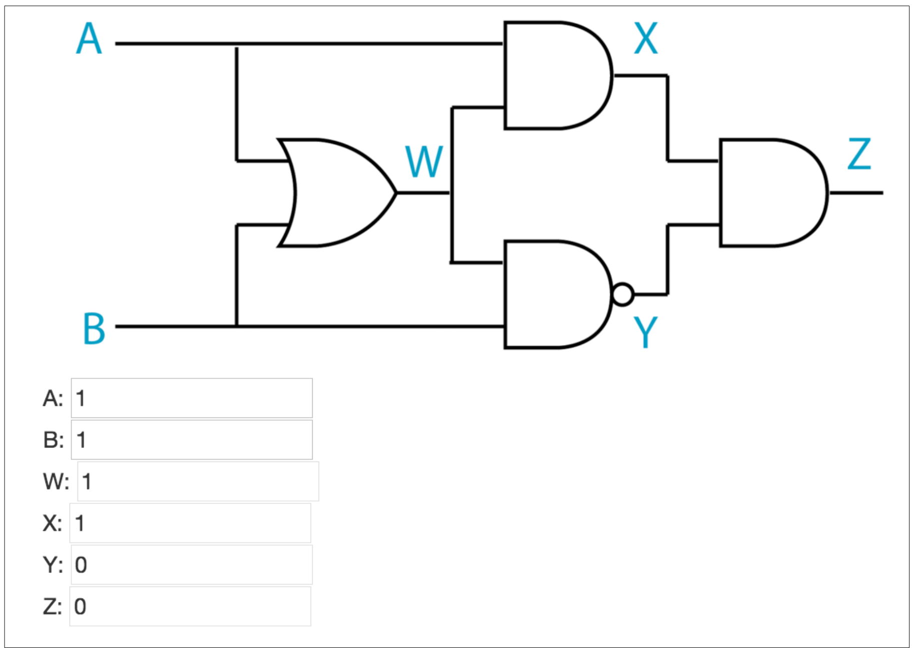

Figure 2: Screenshot of the logic gates user interface

As shown in Figure 2, ”A” and ”B” are the user inputs that takes the binary values of 0 or 1. The results after passing through the various gates are denoted by the letters ”W”,”X”,”Y” and ”Z”.

To start the implementation, we define pure functions to represent the gates. For example a NAND gate as shown in Code example 4.5 below.

```
var nand = function (a, b) {
                if(a == 1 && b == 1)
                    return 0;
                return 1;
            };
```

Code example 4.5: NAND gate implemented as a JavaScript function

These gate functions are pure functions without side-effects and thus we are so far preserving the functional interests of the implementation.

Next, we start defining the data flow of the application by creating event streams for the user inputs. As in the former case, we convert these streams directly to properties after mapping them to the value of the input field. The following example, Code example 4.6, shows one of these two properties.

```
var a = $('#a').asEventStream('keyup')
                .map('.currentTarget.value')
                .toProperty(0);
```

Code example 4.6: Declearing a Bacon property from a input field

With the inputs ready, we can design the data flow through the circuits. For this, we use the same technique as the former case by using Bacon's "combine" function. We combine two properties to a new property *through* a function. The functions we use with the combine function are the gate functions declared before. The result of the data mapping according to Figure 2 is shown below in Code example 4.7.

```
1 var w = a.combine(b, or);
2 var x = a.combine(w, and);
3 var y = b.combine(w, nand);
4 var z = x.combine(y, and);
```

Code example 4.7: Mapping properties together to form new ones through a given function

Note that sending a function as a parameter to the combine function is possible due to JavaScript's support of first-class functions. The new properties now hold the intermediate result values throughout the circuit as illustrated in Figure 2.

Finally, we can assign the value of these properties to the input fields so that they are displayed to the user. This can be achieved by the following code in Code example 4.8.

```
w.assign($('#w'), 'val');
x.assign($('#x'), 'val');
y.assign($('#y'), 'val');
z.assign($('#z'), 'val');
```

Code example 4.8: Assign the properties to respective input fields in the DOM

Nearly all logic behind this implementation is already presented in the examples. As shown, only a small amount of code is needed to handle the logic between the gates and displaying the results on screen. Thus, the developer can implement with a high level of abstraction and does not need to be aware or handle the actual routing of the data. State manipulations of the DOM are also excluded, as we bind the results using the "assign" function. Altogether, this limits the amount of errors and bugs that may be introduced, as the developer has less to handle.

During the research, a pattern similar to what we have seen so far was found useful for doing FRP in a structured way. That is, declaring functions that are to be used by the program first, then design the data flow. When designing data flow, we have used a top to bottom approach that starts with declaring streams/properties, and then transforms, combines, merges, and manipulates data, and lastly displays relevant results to the user.

4.3 Case 3: Real-Time Chat with WebSocket

The goal of this case is to illustrate how FRP works together with a web application using a third party method of communicating across multiple instances. For that purpose we present an example of a web chat that uses a Node.js [25] server to broadcast incoming messages to all other clients using WebSocket [37]. WebSocket is a protocol for two-way communication that is supported in modern web browsers [41] and on the Node platform.

Node.js is a popular framework for doing server-side JavaScript development and allows asynchronous I/O and an event-driven development model. Node takes JavaScript as we know it and execute it on the V8 engine [13] so that it can run on operating systems as a standalone program. That way, there is no need for a browser to execute the code, and the operating system can do this directly through the engine. This means that we can write server-side applications that for example communicates with a database for persisting data, or, in our case, broadcast messages to clients with WebSocket. To utilize WebSocket with Node, one must either implement the communication or use an existing library.

The defacto library for using WebSockets on the web today is Socket.IO (SIO) [33] [34]. SIO features both a server and a client side API to emit and receive data through WebSocket. For this case, we will use SIO as it has grown to be the standard of WebSocket communication on the web, and to implement such functionality from scratch would be tedious (see [34]).

To further make this a practical example, our chat will run on a real web server using Express [9]. Express is a minimalistic web framework for hosting websites and provides features like URL routing.

Following in Code example 4.9 is an example of how an Express application is serving the text ”Hello World!” when visiting http://localhost:3000/ in a browser on your local machine, given that both Node and the Express module is installed.

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function (request, result) {
5    res.send('Hello World!');
6  });
7
8  var server = app.listen(3000, function () {
9    var host = server.address().address;
10   var port = server.address().port;
11 });
```

Code example 4.9: Hello World with Express and Node

Notice that Node is module oriented and that including packages like Express is done elegantly with the ”require” function, in contrast to what we would do in a native client environment by including files. One can easily expand the application with more URL endpoints using the ”app.get()” function as here displayed by the root, ”/”.

With the tools of Node, Express and SIO available, we move on to the implementation of the chat. A user should be able to visit the URL from the Express server and get the user interface of the chat. Upon sending a message, the client's web browser emits the message using SIO to the server. The server then broadcasts this message to all active clients as shown in the following figure, Figure 3.

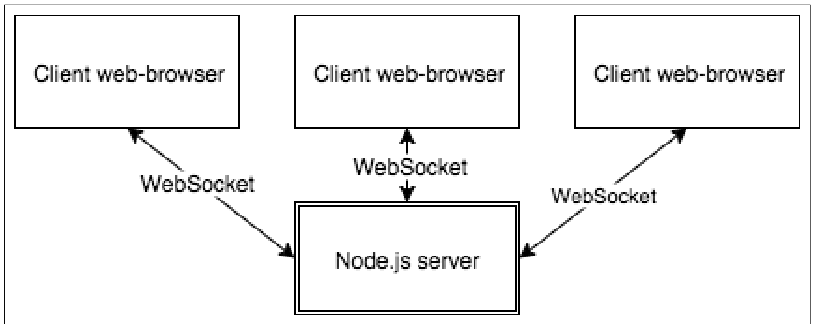

Figure 3: Overview of the chat application's communication flow

First, we will show how the server side is implemented and later on the client side. The skeleton for the app is the same as in the Express example, but instead of returning a text string we render a template. This could be a regular HTML file, but in this example we use a template engine called Jade [16]. Jade allows writing HTML without the use of brackets, and uses indentation for hierarchy. This allows for a less verbose coding style.

To demonstrate some of the difference, consider this example of declaring a script-tag.

```
HTML: <script src="js/chat.js"></script>
```

```
Jade: script(src="js/chat.js")
```

In the server-side code, using such a template engine can be achieved as shown in the next code example, Code example 4.10. In the "app.get()" function, the Jade file "index.jade" will be rendered and sent to the client.

```
var app = express();
app.set('view engine', 'jade');

app.get('/', function (request, result) {
  res.render('index');
});
```

Code example 4.10: Using a template engine with Express

To summarize our previous steps; our application now runs on a URL that can respond with a page when requested. The next step is to allow receiving and emitting messages through the sockets. We import the SIO module (given that it is already installed).

```
var io = require('socket.io').listen(server);
```

SIO is now available through the ”io” variable and is also bound to the server through the ”listen()” function that we passed our running server to as a parameter.

Finally, we listen to incoming messages from our connected clients and broadcast them to all of the sockets the server have in its ”io.sockets” pool, as shown in Code example 4.11.

```
io.sockets.on('connection', function (socket) {
  console.log('Client connected');
  socket.on('message', function (data) {
    io.sockets.emit('message', data);
  });
});
```

Code example 4.11: Listening and broadcasting through WebSockets

This concludes everything needed for the server side of this application. On the client side, we will no longer use Node or Express as they are only required for the server-side. Our web server is already running. Relevant now is what the server sends to the client to interpret. This includes the user interface of the chat, the logic for sending messages to the server, handling received messages, and displaying them on screen. The following figure, Figure 4, shows the web chat in a client's browser with example data from a chat with another client.

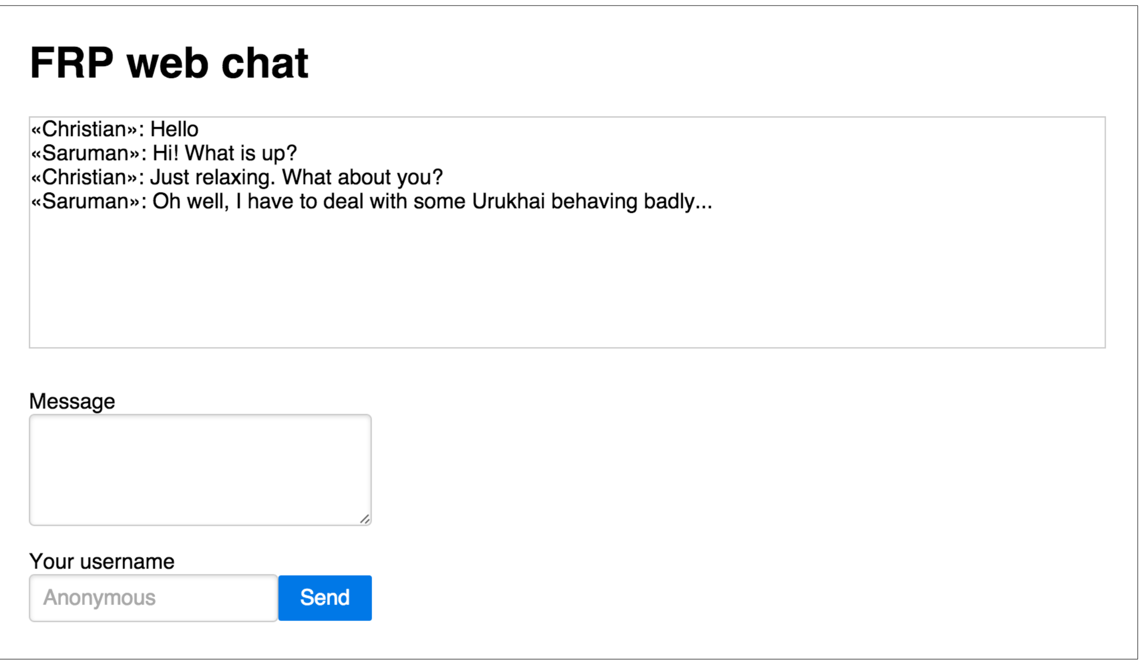

Figure 4: Screenshot of the FRP web chat interface

To send messages to the other clients through our server, we use the client API of the SIO library. This can be achieved in almost the same fashion as with the module in Node:

```
var socket = io.connect();
```

We then apply FRP techniques to capture the data from the user and send it to the socket. To achieve this, we create an event stream from when the user clicks "Send". We map the data on this stream to be a combination of the message and username provided through a "getMessageAndUsername" function, before declaring, that, when a value is pushed to the stream, we should emit it to the socket. This is shown in Code example 4.12.

```
var messageStream = $('#sender')
                    .asEventStream('click')
                    .map(getMessageAndUsername)
                    .onValue(function (data) {
                        socket.emit('message', data);
                    });
```

Code example 4.12: Event stream that emits a message to socket

When the message is received by the socket on the web server, it is broadcasted to the other clients as shown in the server-code example, Code example 4.11. Upon arrival on the client, we must receive the message and display it on the screen of the client. Consider Code example 4.13 first, and we will explain how this can be done using Bacon and FRP.

```
var fromServer =  Bacon
                    .fromEventTarget(socket, 'message')
                    .map(constructMessage)
                    .toProperty()
                    .assign($('#chat'), 'append');
```

Code example 4.13: Event stream that emits a message to socket

Familiar functions like "map", "toProperty", and "assign" are present, but a different technique is required to retrieve the message from the socket. For this we use Bacon's "fromEventTarget" function. This function creates a new event stream from a DOM EventTarget, e.g., a click on an element, or a Node.js EventEmitter object, such as the socket from Socket.IO. In practice,

we can convert the data on the socket to a familiar event stream and from there apply all FRP techniques and methods we have shown previously as needed.

From the event stream, we map the data through the ”constructMessage” method. The data coming from the socket is in the form of a JavaScript object:

```
{message: 'Hello', username: 'Saruman'}
```

The ”constructMessage” method simply parses this data to a more readable format for display to the user. We then convert the message to a property so that its value can be assigned to an element in the DOM and thus will appear in the client's web browser.

An experienced developer might argue that receiving and sending messages in the manner we illustrate could easily be done in a traditional imperative way with simple event listeners. Although this is true, this case shows an example of how Bacon and FRP integrate with third-party libraries and technologies like Node and WebSocket/Socket.IO. Abstracting the socket layer by transforming the sockets to event streams opens up for adding complexity while preserving the ideas of the paradigm. An imperative approach might not handle as well when complexity and features are added.

4.4 Case: Complex Client Side Form Validation

The Form is an essential component of the web. Traditionally, a form consists of some input elements and a submit button. When submitting the form, a specified action is executed. A common approach is to make the form do an HTTP POST request [15] to a script, e.g., a PHP script [26] that handles the data from the form passed on from the request. For example, when registering a user on a web site. The user fills out his credentials as specified by the form and presses submit. The form sends a POST request to a script that persists the user's data in a database.

Using the built in mechanics of a form like in the above example forces a page reload and thus a request-response cycle to the server. If something went wrong and the form is not valid, the server must detect this and provide feedback to the user. This is inefficient. A better approach is to use JavaScript to validate the form on the client-side before sending it to the server and thus relieving the server of load.

Validating and providing the user with intuitive feedback throughout the form can quickly become complex as the different input fields might depend on each other and force strict rules. Delivering useful feedback to the user on each field also means that we have to manipulate the DOM, i.e., insert messages consecutively on the site in the relevant parts.

A modern form should be able to provide instant feedback throughout multiple fields with multiple rules attached to them. Fields can even be dependent of each other, and have external communication, such as lookups in a database. The challenges we have discussed so far implies that an implementation of such a form will be tough and touch by many of the problems we may encounter by doing this in an imperative way.

With that in mind, we move forward to the specification of the case. A seemingly good candidate for FRP, as a form may have complex state and

DOM manipulation with several input sources and internal dependencies.

The form in this case is a registration form for users on a website. We focus on the client side validation, and nothing on the server. Note that it is important to have both client and server side validation, because the request with the data that is sent to the server eventually can be forged and thus bypass our rules. The aim of client-side validation is to relieve the server of request cycles during the validation process, and giving the user a fluent experience while filling out the form. Implementing validation on both client and server-side means more work, but the server validation does not need to provide thorough feedback like the client-side and is merely an extra security layer.

The form consists of the following fields, each with their corresponding rules. Details regarding the rules will be discussed later.

- **Username:** The username field has an external resource lookup to ensure the user that the username is available. In addition, the username cannot be empty, and only letters and numbers are allowed.
- **Full name:** The full name of the user, each name must be capitalized, and the field cannot be empty.
- **Password:** A password must be at least six characters long.
- **Confirm password:** This field must contain the exact same characters as the "Password" field.
- **Register button:** This button will only be enabled as clickable if all the previous fields are valid.

The user interface including what we have specified is displayed in Figure 5 below. To ease the design process of scaffolding and styling the elements, as that is not the focus of this thesis, we use the component based front-end framework, Twitter Bootstrap [38]. Bootstrap does not have any impact on the dynamic JavaScript of our implementation and is only used to style the HTML elements.

Figure 5: Screenshot of the user interface of the form

Using the same approach as in the previous cases, we start with defining functions that are used when modeling the data flow. For each rule we have defined, we create a corresponding validation function as shown in the next example, Code example 4.14. Such a function takes input that comes from the input fields' respective streams. This input is then validated, and if passed, the value itself is returned so that it can pass through our streams further on. If, however, the validation fails, a "Bacon.Error" object is returned with an error message attached. We will see later how we can filter out these errors and display them to the user.

```
var passwordLongEnough = function (password) {
    var errorMessage = 'Password must be atleast six characters long';
    if(password.length < 6)
        return new Bacon.Error(errorMessage);
    return password;
};
```

Code example 4.14: One of the validation functions for the FRP form

With the logic of our rules ready, we move on to the top of our data flow. That is, streams for each input field. Our form has four input fields that is our data foundation for further use. We declare streams that will yield the value of the field in the form of a property for each of the input fields, as shown in code example 4.15 below. To avoid code redundancy in the stream creation, we introduce a help function.

```
var createFieldPropertyStream = function ($element) {
    return $element
            .asEventStream('keyup')
            .map('.currentTarget.value')
            .toProperty('');
};

var fullname = createFieldPropertyStream($('#fullname'));
```

Code example 4.15: Creating the properties for the input fields

The data foundation is now available in related property variables as the "fullname" variable declared in the example. Attaching the validation

functions to our pipe from the input can be done in the following way as exemplified in the field ”Full name”.

```
var validFullname = fullname.flatMap(nonEmpty).flatMap(checkCapitalizedNames);
```

”validFullname” is now a new property chained by the functions ”nonEmpty” and ”checkCapitalizedNames” using the flatMap technique from Bacon. FlatMap is one of the more complex but rewarding concepts to understand in FRP. It is an operation different from the regular map function we have used so far. While map will only apply a function to the data yielded by a stream, thus replacing the original value with a transformed one, flatMap will flatten the stream as well. To explain the flattening procedure, consider the following structure with two levels.

```
[ [1,2,3], [4,5,6] ]
```

When flattening, we reduce this to one level.

```
[ 1,2,3,4,5,6 ]
```

In context, this means we spawn a new stream for the new values of the given parameter to flatMap and concatenate it with the original stream forming a result stream. In our case, this result stream is ”validFullname”. This result stream, which is a property in our case since we declared it as such in code example 4.14, will hold either the value of the input field or a Bacon.Error object if the validation failed.

This technique using flatMap for the validation functions can be applied to solve all such instances in the form, except checking user availability and confirming password equality.

To perform checking if a username is available or not from an external resource, we have mocked this with a simple PHP API (Application Programming Interface). The API is an endpoint that can be requested by

HTTP GET with a set parameter of the username, and will yield a response that can tell us if it is available or not. To be able to request this API without manually entering the URL in the browser, or force page reloads, we introduce AJAX (Asynchronous JavaScript and XML), [1]. AJAX is the use of the XMLHttpRequest object to communicate with server-side scripts asynchronously. The jQuery library [18] we utilize in this thesis to simplify our code includes an elegant implementation of AJAX for easy use. Because of the asynchronicity, waiting for a response is non-blocking, and we use a success callback to verify that we are not handling data from the response before its arrival. Following is an example of an AJAX request in jQuery.

```
1 $.GET('http://example.org/someEndpoint/')
2     .success(function (response) {
3         console.log(response);
4     });
```

Code example 4.16: Example of an AJAX request using HTTP GET

To use the PHP API with our Form and FRP, we convert it to a stream by using ”fromPromise” function from Bacon. This function will work on other promise implementations like jQuery Deferred [17]. A promise object is a pattern or technique used in asynchronous communication that will deliver a *promise* of incoming data, but does not know when the data will arrive. We can look at it as a temporary object that will be resolved to an object of meaning once the data arrives. We use this technique to program with the responses from AJAX.

In our application, we create a function that wraps the stream conversion from our AJAX request in code example 4.16 below.

```
var ajaxUsernameStream = function (query) {
      return Bacon.fromPromise($.get('api.php?username=' + query));
  };
```

Code example 4.17: Converting AJAX to a Bacon stream

In the chain of validating a user, we include the AJAX stream and end up with the following result as shown in code example 4.17.

```
var validUsername = username
              .flatMap(nonEmpty)
              .flatMap(checkValidUsernameChars)
              .combine(
                  username
                    .flatMapLatest(ajaxUsernameStream)
                    .flatMap(usernameNotTaken),
                  function (validity, available) {
                    if (available === true) return validity;
                        return available;
                  }
              );
```

Code example 4.18: The chain of validating the username

The code shown in the example can be rather hard to grasp, as there is much going on at once. We start with a similar technique as before with flat mapping validation functions. These first three lines are so far flattened to one property. We then take this property and combine it with the stream

consisting of the original username stream, flat mapped with the username availabilty check, through the anonymous function starting at line 8.

As the AJAX request is indeed asynchronous, we must take into account that an older response might return before a newer one. For that purpose, we use the ”flatMapLatest” function from Bacon to ensure our data is up to date. The AJAX stream is flat mapped with the function ”usernameNotTaken” which is used to handle errors on the stream. Finally, we combine through the anonymous function, so that if the username is available, we return the result from the property consisting of the first three lines. Otherwise, the property will contain the error yielded by the ”usernameNotTaken” function. We recommend reviewing Appendix 4 for the complete code when trying to understand the mapping of data here. The combination of observables, i.e., streams, and properties, in code example 4.17 was one of the most challenging tasks to implement in this thesis. It is possible that it could be done more elegantly.

Our last input field is for confirming the password. Since all the validation for the password itself is handled by the password field, there is no need to duplicate this as the form would not be valid if either field contained errors. For this to be valid, it only has to contain the exact same text as the password field. The validation is shown in code example 4.18 below.

```
var passwordsEqual = password
                .combine(passwordConfirm,
                    function (pass, confirm) {
                        if(pass === confirm)
                            return pass;
                        return false;
                    }
                )
                .flatMap(passwordsAreEqual);
```

Code example 4.19: Validation of matching passwords

As before, we combine the value of both fields and check if their respective strings are equal. To handle errors, we flat map onto the chain a "passwordsAreEqual" function that will either return a Bacon.Error or the password.

To summarize, we now have observable properties for each of our fields that either contains errors or a vaild value. We now move on to display eventual errors to the user. For this purpose we have implemented a help function which method signature takes an element and an error message:

```
var displayError = function ($wrapper, errorMessage)
```

This function will insert or remove a help-block according to the error handling component in our Bootstrap framework. Such errors can be viewed in Figure 5, shown previously in this chapter, and consists of a red border around the input field and a corresponding red error message below. We also have a "removeError" function which does the exact opposite.

The concept is to check the properties for errors. If they occur, display the error, if not, remove any previous errors. We must also control what

feedback belongs to what field. For that purpose we use the ”bindFeedback” function in Code example 4.19 below.

```
1  var bindFeedback = function (stream, $wrapper) {
2      stream
3        .onError(displayError.bind(null, $wrapper));
4      stream
5        .skipErrors()
6        .skipDuplicates()
7        .onValue(removeError.bind(null, $wrapper));
8  };
9
10 bindFeedback(validFullname, $fullname);
```

Code example 4.20: Binding feedback to fields to the correct element

As we see an example of on line 10, we pass the element that wraps the corresponding input field so that we can use it in the ”onError/onValue” functions. The technique used is JavaScript’s bind function, [21]. This function will create a new function, that when called, has the value of the wrapper already set. The result is that when any of the properties contains a Bacon.Error, ”displayError” will be called. If a field is valid, existing error feedback will be removed. Each field is bound like this independantly by the ”bindFeedback” function. The result is a working form that provide real-time feedback to its respective field depending on of what the user enters.

All our validation properties can also be combined to tell when the whole form is valid, and thus when we can enable the register button. This is shown in code example 4.20 below.

```
var isValid = validUsername
            .combine(validFullname, '.concat')
            .combine(validPassword, '.concat')
            .combine(passwordsEqual, '.concat')
            .map(false).mapError(true);

isValid.assign($('button'), "attr", "disabled");
```

Code example 4.21: Binding feedback on fields to the correct element

In this example we create a new property by combining all other properties through the predefined ".concat" function. We then map the result to false if any the chained functions contains an error, or true if not. We can then assign this property to control the "disabled" attribute on the button so that if the form is not valid, the user will not be able to submit the form.

5 Discussion

This chapter will discuss the findings from the previous chapter containing the four case implementations with regards to the research questions from Chapter 3.4. The aim is to shed light on how FRP applies on the web, and if this change in paradigm is sustainable for the future.

We start with discussing the code quality of our implementations. Although this is arguably a topic of a personal manner, there is research established that agrees on what is definitive bad code in the context of JavaScript, as shown in [10]. For example, excessive nesting of callbacks to ensure concurrency, or weak implementations of manual state handling. Some of these concepts are by design difficult to do wrong when using FRP. Some of the motivation behind reactive programming on the web is to counter these pitfalls that JavaScript allows so easily.

In Case 1 we compared the reactive approach to an imperative one. Although the reactive version had less code and more functionality, the overall amount of code needed to achieve the same functionality was roughly the same. Although this thesis does not include side-by-side comparisons with imperative, or similar solutions, to show more complex cases, the implementations shown are fully working with basic functionality. None of our findings trigger any alarms of serious deviations of code size, but rather a somewhat equal amount as shown in Case 1.

The pattern in which code is written in our implementations with a top-down approach might be preferable for developers as readability, i.e., the ability to follow the code and intuitively understand what it does, increases. Imperative solutions may have scattered parts of the implementation in various sections due to the asynchronicity and need for callbacks to handle application concurrency management.

The implementation we have used (Bacon), makes it fairly easy to stay

within the boundaries of the paradigm, at least, when modeling data-flow. One does not have the need to introduce, e.g., callbacks, because Bacon provides tools to solve the problems in a reactive manner. With our top-down approach, it is up to the developer to maintain the ideology of the paradigm when introducing logic functions or help functions (e.g., Code example 4.4), as these are not part of the Bacon API. This is where the inexperienced might be tempted to fall back to old habits, as there is no safety net from the library. In Case 4, some of our help functions manipulates the DOM to give the users feedback in the form. This was done because we could not find any way of doing this with pure Bacon code. In the case, only small pieces of code are extracted into semantic functions that are controlled by reactive components. This does not necessarily imply bad design, but as the feature set increases, the developer must keep these issues in mind to ensure a clean semantic structure. The findings in the results of this thesis did not uncover a way to mitigate this problem.

Hand in hand with code quality comes maintainability. It is common today that multiple people work on the same projects, and thus touch and enhances each other's code. There are many costs tied to not being able to understand what someone else wrote, and from a business perspective, such costs can bring an organization to its knees. Therefore, developers today must strive to write clean and maintainable code. This is known as software craftsmanship, [19]. This is relevant as this thesis is a practical evaluation. We aim to evaluate how our findings apply to the real world. When looking back at our cases, we can see promising results even in the first example, Code example 4.3, regarding simple addition. Further validation of the input fields is needed to make a solid implementation that ensures that a result can be calculated. We provide an example of this by mapping on other functions, like ”isNumber” to verify that the ”parseInt” function did not return NotaNumber(NaN). This is an example of enhancement. Further

on, in Case 4, we can see that expanding validations of an input field can be done elegantly by implementing more validation functions and chaining them by using the ”flatMap” function. Even adding another input field would not have any substantial impact on any of the code already written, and could easily follow the same pattern: Convert input to event stream, flatMap validation functions onto stream, bind feedback to user.

This thesis only shows how small enhancements or, how adding similar features to what already exists in the application can be done. Further work to measure maintainability would include a major change to the specifications of our implementations. Unfortunately, this was out of scope of the thesis.

The limitations of FRP in our thesis will be mostly narrowed to the context of our Bacon library. The API provided by the library has a set of functions that are already defined and cannot be altered by the developer (without modifying the library). These functions, such as ”asEventStream” and ”flatMap”, are documented tools for the developer to wield and combine to build an application. As previously discussed regarding Case 4, there may be problems that cannot be solved by using Bacon alone. Assigning values to existing elements in the DOM work seamlessly, but there may be the need to alter the DOM itself. There are other libraries and frameworks, like React by Facebook [28] that operates with a virtual DOM for this purpose, and it is possible that such techniques could combine with FRP. This is unfortunately out of scope for this thesis, but can be considered for future work.

The Bacon library illustrates a real promise for integrating with external resources, as shown both in Case 3 with the chat, and Case 4 with the external API for username availability. The concept of converting such resources to the familiar event-stream makes it easy to maintain a level of abstraction towards other services or resources not directly coupled with the application in question. This aids the developer in concentrating on relevant parts and increases the maintainability factor with separation of concerns [19].

One aspect that is relevant to discuss in the context of possible multiple developers in an organization is adoption. As we stand today, the FRP paradigm might be completely new to a developer. The results, as shown in the implementation, may show real promise for the future of FRP on the web, but the way along may be tedious. Some of the concepts we have shown can be hard to grasp for the first time for a developer schooled in imperative or object oriented paradigms, either from education or experiences from previous projects.

Introducing FRP in a new environment comes at a cost. The implementation of FRP in the form of a library includes code that must be delivered to the client along with the application itself, for example over the Internet. Although Bacon, in our case, is relatively small, when working with client-side code, it is good practice to keep transfer sizes as small as possible to ensure performance in the application as developers do not know the specifications of the client's hardware.

Also, when problems arise in an existing application that seem fit for FRP, inclusion of a new library featuring a whole new paradigm must be carefully considered. The positive impacts of FRP might not shine through the cost of introducing a new paradigm for the developers to maintain, as well as the performance hit by adding another library.

6 Conclusion

The web as an application platform is growing rapidly. As JavaScript both on the server and client side is treasured by developers to build complex solutions for the web, we face different challenges. The forgiving nature of JavaScript allows emerging of inadequate coding styles and a lack of discipline when creating software. Most of today's JavaScript applications are written in an imperative manner, that when combined with event-based programming to handle concurrency management in an asynchronous environment leads to the introduction of poor practices. The Functional Reactive Programming (FRP) paradigm is a different approach to doing such asynchronous development. Implementations meant for the web, e.g., Bacon as used in this thesis, counters the problems and challenges from traditional ways reviewed in this thesis.

In this thesis, we have used a case based practical approach for evaluating how FRP can apply to the web. Through four different problems and applications, we have implemented different specifications of such problems using an FRP implementation of JavaScript, namely Bacon. These cases demonstrate both the power of reactive programming when e.g., managing state, handling concurrency and working with internal dependencies in an application. They also demonstrate how we apply ideas from the functional paradigm and combine these with reactivity to ensure good extraction that leads to readability for developers. Techniques from the functional paradigm are also utilized by Bacon to transform data directly on our various observable data types.

The implementations use a pattern when designing code inspired by the classic FRP from the 90s that feature a top-down approach where functions used when modeling are declared first. Then the data flow is modeled in a top-down way, where we go from declaring our sources from inputs to combining and transforming the data to create a data foundation. Finally, the data

foundation is presented to the user using minimal DOM manipulation from Bacon's minimalistic DOM-related functions, which e.g., assigns properties to attributes on elements..

The findings from our results show that a broad selection of common problems faced in an average organization can be solved with FRP. The implementations do not deviate from standards regarding code size, and readability and maintainability may improve drastically by using FRP. Expansion and enhancement of the applications can be done elegantly as discussed when they do not involve drastic alterations of the specification. Further work should include testing and measuring this on a larger scale, as this thesis is merely an introduction to applying FRP in practice and has a limited scope.

Although FRP shows great promise from our findings, a developer inexperienced in the reactive ways may have a hard time grasping the new concepts. Organizations and businesses must carefully consider the cost of introducing a new paradigm for this reason, and others discussed in this thesis such as performance hits, and maintainability. Our conclusions lean towards promise when building FRP applications from scratch, and see more challenges with enforcing FRP in existing environments. Leading organizations such as Netflix at the time of writing have had great success with converting their entire frontend platform using similar techniques that we have used in this thesis.

Thus, we conclude that FRP is a good candidate for consideration when building complex, stateful applications with several input sources.

References

[1] *AJAX: W3C XmlHTTPRequest Level 2*. URL: `http://www.w3.org/TR/2012/WD-XMLHttpRequest-20120117/`.

[2] *Apple Mac OSX, version Yosemite*. URL: `http://www.apple.com/osx/`.

[3] *Bacon.js download count sampled 25-11-2015 from NPM Stat*. URL: `http://npm-stat.com/charts.html?package=baconjs&author=&from=2013-11-25&to=2015-11-25`.

[4] *Bacon.js Functional Reactive Programming library for JavaScript*. URL: `https://github.com/baconjs/bacon.js`.

[5] Evan Czaplicki. "Elm: Concurrent FRP for Functional GUIs". In: *Senior thesis, Harvard University* (2012).

[6] Christoph Doblander, Thomas Parsch, and Hans-Arno Jacobsen. "Geospatial event analytics leveraging reactive programming". In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM. 2015, pp. 324–325.

[7] ECMA ECMAScript, European Computer Manufacturers Association, et al. *ECMAScript Language Specification*. 2011.

[8] Conal Elliott and Paul Hudak. "Functional reactive animation". In: *ACM SIGPLAN Notices*. Vol. 32. 8. ACM. 1997, pp. 263–273.

[9] *Express, Node.js web framework*. URL: `http://expressjs.com/en/index.html`.

[10] Amin Milani Fard and Ali Mesbah. "JSNOSE: Detecting JavaScript code smells". In: *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE. 2013, pp. 116–125.

[11] David Flanagan. *JavaScript: the definitive guide.* " O'Reilly Media, Inc.", 2006.

[12] *Google Chrome web browser.* URL: https://www.google.com/chrome/.

[13] *Google, V8 JavaScript engine version 5.* URL: https://code.google.com/p/v8/.

[14] *Haskell programming language.* URL: https://www.haskell.org/.

[15] *HTTP Protocol RFC.* URL: https://tools.ietf.org/html/rfc2616.

[16] *Jade Template Engine.* URL: http://jade-lang.com/.

[17] *jQuery Deferred Promise.* URL: https://api.jquery.com/deferred.promise/.

[18] *jQuery JavaScript library version 2.1.3.* URL: https://jquery.com/.

[19] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[20] D.S. McFarland. *CSS3: The Missing Manual.* Missing manual. O'Reilly Media, 2012. ISBN: 9781449339494. URL: https://books.google.no/books?id=cqeTt3RhKF0C.

[21] *MDN: JavaScript prototype bind.* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind.

[22] *Module Counts, sample from 04-11-2015.* URL: http://www.modulecounts.com/.

[23] *Netflix Q3 2015 letter to shareholders.* URL: http://files.shareholder.com/downloads/NFLX/1064918284x0x854558/9B28F30F-BF2F-4C5D-AAFF-AA9AA8F4779D/FINAL_Q3_15_Letter_to_Shareholders_With_Tables_.pdf.

[24] Henrik Nilsson, Antony Courtney, and John Peterson. "Functional reactive programming, continued". In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM. 2002, pp. 51–64.

[25] *Node.js v5.1.0*. URL: `https://nodejs.org/en/`.

[26] *PHP programming language*. URL: `http://php.net`.

[27] Mark Pilgrim. *HTML5: up and running*. " O'Reilly Media, Inc.", 2010.

[28] *React JavaScript library for building user interfaces*. URL: `https://facebook.github.io/react/`.

[29] *Reactive Banana, Haskell FRP library*. URL: `https://wiki.haskell.org/Reactive-banana`.

[30] *Reactive Extensions for JavaScript by Microsoft*. URL: `https://github.com/Reactive-Extensions/RxJS`.

[31] *Reactive Programming in Netflix - Netflix Blog*. URL: `http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html`.

[32] *Rx download count sampled 25-11-2015 from NPM Stat*. URL: `http://npm-stat.com/charts.html?package=rx&author=&from=&to=`.

[33] *Socket.IO library for WebSocket*. URL: `http://socket.io/`.

[34] Pedro Teixeira. *Professional Node. js: Building JavaScript based scalable software*. John Wiley & Sons, 2012.

[35] *The Python Programming Language*. URL: `https://www.python.org/`.

[36] *The Reactive Exensions project by Microsoft*. URL: `https://msdn.microsoft.com/en-us/data/gg577609`.

[37] *The WebSocket Protocol*. URL: `https://tools.ietf.org/html/rfc6455`.

[38] *Twitter Bootstrap v3*. URL: `http://getbootstrap.com/`.

[39] *W3Schools: Anonyomous functions in JavaScript*. URL: http://www.w3schools.com/js/js_function_definition.asp.

[40] Zhanyong Wan, Walid Taha, and Paul Hudak. "Real-time FRP". In: *ACM SIGPLAN Notices*. Vol. 36. 10. ACM. 2001, pp. 146–156.

[41] *WebSocket browser support, sampled 01-12-2015*. URL: http://caniuse.com/#search=websocket.

A Appendix 1: Simple Addition

A.1 Simple Addition using Bacon and FRP

```
<!DOCTYPE html>
<html>
    <head>
        <script
        ↪ src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/
            libs/bacon.js/0.7.53/Bacon.min.js">
        </script>
    </head>
    <body>
        <h3>Simple addition</h3>
        <input type="text" id="a"> +
        <input type="text" id="b"> =
        <input type="text" id="c">

        <script>
            (function () {
                var isNumber = function (input) {
                    return input > 0 || input < 0;
                };
                var sum = function (x, y) {
                    return x + y;
                };

                var a = $('#a')
                        .asEventStream('keyup')
                        .map('.currentTarget.value')
                        .map(parseInt)
                        .filter(isNumber)
                        .toProperty(0);

                var b = $('#b')
```

```
32                          .asEventStream('keyup')
33                          .map('.currentTarget.value')
34                          .map(parseInt)
35                          .filter(isNumber)
36                          .toProperty(0);
37
38                  var c = a.combine(b, sum);
39                  c.assign($('#c'), 'val');
40              }());
41          </script>
42      </body>
43  </html>
```

A.2 Simple Addition with jQuery

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <script
            ↪   src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
5           <script src="https://cdnjs.cloudflare.com/ajax/
6               libs/bacon.js/0.7.53/Bacon.min.js">
7           </script>
8       </head>
9       <body>
10          <input type="text" id="a" value="0"> +
11          <input type="text" id="b"> =
12          <input type="text" id="c">
13
14          <script>
15              $(function () {
16                  $('#a').on('keyup', function () {
17                      var a = parseInt($('#a').val());
18                      var b = parseInt($('#b').val());
```

```
19                     $('#c').val(a + b);
20                 });
21 
22                 $('#b').on('keyup', function () {
23                     var a = parseInt($('#a').val());
24                     var b = parseInt($('#b').val());
25                     $('#c').val(a + b);
26                 });
27             });
28         </script>
29     </body>
30 </html>
```

A Appendix 2: Logic Gates

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <link rel="stylesheet"
           ↪  href="https://maxcdn.bootstrapcdn.com/bootstrap
5              /3.3.5/css/bootstrap.min.css" type="text/css">
6          <script
           ↪  src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
7          <script src="https://cdnjs.cloudflare.com/ajax/
8              libs/bacon.js/0.7.53/Bacon.min.js"></script>
9      </head>
10     <body>
11         <div class="container">
12             <div class="col-md-6">
13                 <img src="./logic-gates.png" alt=""
                   ↪  width="100%">
14                 A: <input type="text" id="a"><br />
15                 B: <input type="text" id="b"><br />
16                 W: <input type="text" id="w"
                   ↪  disabled="true"><br />
17                 X: <input type="text" id="x"
                   ↪  disabled="true"><br />
18                 Y: <input type="text" id="y"
                   ↪  disabled="true"><br />
19                 Z: <input type="text" id="z"
                   ↪  disabled="true"><br />
20             </div>
21         </div>
22
23         <script>
24             (function () {
25
26                 // Logic gates
27                 var or = function (a, b) {
```

```
28                  if(a == 1 || b == 1)
29                      return 1;
30                  return 0;
31              };
32              var and = function (a, b) {
33                  if(a == 1 && b == 1)
34                      return 1;
35                  return 0;
36              };
37              var nand = function (a, b) {
38                  if(a == 1 && b == 1)
39                      return 0;
40                  return 1;
41              };
42
43              // Data flow
44              var a = $('#a')
45                  .asEventStream('keyup')
46                  .map('.currentTarget.value')
47                  .toProperty(0);
48
49              var b = $('#b')
50                  .asEventStream('keyup')
51                  .map('.currentTarget.value')
52                  .toProperty(0);
53
54              var w = a.combine(b, or);
55              var x = a.combine(w, and);
56              var y = b.combine(w, nand);
57              var z = x.combine(y, and);
58
59              w.assign($('#w'), 'val');
60              x.assign($('#x'), 'val');
61              y.assign($('#y'), 'val');
62              z.assign($('#z'), 'val');
63          }());
64      </script>
65  </body>
```

```
66  </html>
```

A Appendix 3: Real-Time Chat with WebSocket

A.1 Server-side Node

```
var express = require('express');

// App setup
var app = express();
app.set('view engine', 'jade');
app.use(express.static(__dirname + '/assets'));

// Routes
app.get('/', function (req, res) {
  res.render('index');
});

var server = app.listen(3000);
var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
    console.log('Client connected');
  socket.on('message', function (data) {
    io.sockets.emit('message', data);
  });
});
```

A.2 Client-side Bacon

```
(function () {
    var socket = io.connect();

    var emitMessage = function (data) {
        socket.emit('message', data);
    }

    var getMessageAndUsername = function (event) {
        var message =
        {
          'message': $('#message').val(),
          'username': $('#username').val()
        };
        return message;
    }

    var constructMessage = function (message) {
        return "&laquo;" + message.username + "&raquo;: " +
         ↪  message.message + "<br />";
    }


    var messageStream =  $('#sender')
                        .asEventStream('click')
                        .map(getMessageAndUsername)
                        .onValue(function (data) {
                            socket.emit('message', data);
                        });

    var fromServer =  Bacon
                        .fromEventTarget(socket, 'message')
                        .map(constructMessage)
                        .toProperty()
                        .assign($('#chat'), 'append');
```

```
34 }());
```

A.3 Client-side Jade Markup

```
1  doctype html
2  html
3    head
4      link(rel="stylesheet",
       ↪  href="http://yui.yahooapis.com/pure/0.6.0/pure-min.css")
5
6      style.
7        .l-box {
8          padding: 1em;
9        }
10
11       #chat {
12         border: 1px solid #CCC;
13         min-height:300px;
14       }
15   body
16     .purge-g
17       .pure-u-1-2
18         .l-box
19           h1 FRP web chat
20           #chat
21           #messagecount(hidden)
22       .pure-u-1
23         .l-box
24           .pure-form
25             label(for="message") Message
26             br
27             textarea#message(type="text", val="Anonymous")
28             br
29             br
```

```
30              label(for="username") Your username
31              br
32              input#username(type="text",
                 ↪  placeholder="Anonymous")
33              button#sender.pure-button.pure-button-primary
                 ↪  Send
34
35      script(src="https://ajax.googleapis.com/ajax/
36          libs/jquery/2.1.3/jquery.min.js")
37      script(src="https://cdnjs.cloudflare.com/ajax/
38          libs/bacon.js/0.7.65/Bacon.min.js")
39      script(src="https://cdn.socket.io/socket.io-1.3.5.js")
40      script(src="js/chat.js")
```

A Appendix 4: Complex Client Side Form Validation

A.1 Form Implementation with Bacon

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <link rel="stylesheet"
       ↪  href="https://maxcdn.bootstrapcdn.com/bootstrap/
5          3.3.5/css/bootstrap.min.css" type="text/css">
6      <script
       ↪  src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
7      <script src="https://cdnjs.cloudflare.com/ajax/
8          libs/bacon.js/0.7.53/Bacon.min.js"></script>
9  </head>
10 <body>
11     <div class="container">
12         <div class="col-md-12">
13             <form>
14                 <div id="frpform-username"
                   ↪  class="form-group">
15                     <label
                       ↪  for="username-id">Username</label>
16                     <input type="text" class="form-control"
                       ↪  id="username-id"
                       ↪  placeholder="Username">
17                 </div>
18                 <div id="frpform-fullname"
                   ↪  class="form-group has-feedback">
19                     <label for="username">Full name</label>
20                     <input type="text" class="form-control"
                       ↪  id="fullname" placeholder="Full
                       ↪  name">
21                 </div>
```

```
22                  <div id="frpform-password"
                     ↪  class="form-group has-feedback">
23                      <label for="username">Password</label>
24                      <input type="password"
                         ↪  class="form-control" id="password"
                         ↪  placeholder="Password">
25                  </div>
26                  <div id="frpform-password-confirm"
                     ↪  class="form-group has-feedback">
27                      <label for="username">Confirm
                         ↪  password</label>
28                      <input type="password"
                         ↪  class="form-control"
                         ↪  id="password-confirm"
                         ↪  placeholder="Password">
29                  </div>
30                  <button class="btn btn-primary"
                     ↪  disabled="true">Register</button>
31              </form>
32          </div>
33      </div>
34
35      <script>
36          (function () {
37            var createFieldPropertyStream = function
               ↪  ($element) {
38                return $element
39                  .asEventStream('keyup')
40                  .map('.currentTarget.value')
41                  .toProperty('');
42            };
43            var displayError = function ($wrapper,
               ↪  errorMessage) {
44                var $existing = $wrapper.find('.help-block');
45                var errorHtml = '<span class="help-block">' +
                   ↪  errorMessage + '</span>';
46                if ($existing.length) {
47                  $existing.html(errorMessage);
```

```
48                } else {
49                  $wrapper.append(errorHtml);
50                }
51                $wrapper.removeClass('has-sucess');
52                $wrapper.addClass('has-error');
53            };
54            var removeError = function ($wrapper) {
55              $wrapper.find('.help-block').remove();
56              $wrapper.addClass('has-sucess');
57              $wrapper.removeClass('has-error');
58            };
59            var ajaxUsernameStream = function (query) {
60                return
                ↪  Bacon.fromPromise($.get('api.php?username='
                ↪  + query));
61            };
62            // Validation functions
63            var nonEmpty = function (value) {
64                return (!value.length > 0) ?
65                  new Bacon.Error('Field cannot be empty') :
66                  value;
67            };
68            var checkCapitalizedNames = function (name) {
69              if (!name.split) return name;
70              var names = name.split(' ');
71              for(var i = 0; i < names.length; i++)
72                  if(!/^[A-Z]/.test(names[i]))
73                      return new Bacon.Error('All names must
                      ↪  be capitalized');
74              return name;
75            };
76            var checkValidUsernameChars = function (username)
            ↪  {
77                if(/^[a-zA-Z0-9]+$/.test(username))
78                    return username;
79                return new Bacon.Error('Username can only
                ↪  contain letters and numbers');
80            };
```

```
81          var usernameNotTaken = function
             ↪  (usernameFromApiAvailable) {
82              if(usernameFromApiAvailable === 'true')
83                  return true;
84              return new Bacon.Error('Username already
                 ↪  taken');
85          };
86          var passwordLongEnough = function (password) {
87            if(password.length < 6)
88                return new Bacon.Error('Password must be
                   ↪  atleast six characters long');
89            return password;
90          };
91          var passwordsAreEqual = function (passwordEqual)
             ↪  {
92            if(passwordEqual == false)
93                return new Bacon.Error('Passwords does not
                   ↪  match');
94            return passwordEqual;
95          };
96          var bindFeedback = function (stream, $wrapper) {
97            stream
98              .onError(displayError.bind(null, $wrapper));
99            stream
100             .skipErrors()
101             .skipDuplicates()
102             .onValue(removeError.bind(null, $wrapper));
103         };
104         // Field wrappers
105         var $username = $('#frpform-username');
106         var $fullname = $('#frpform-fullname');
107         var $password = $('#frpform-password');
108         var $passwordConfirm =
             ↪  $('#frpform-password-confirm');
109
110         // Streams
111         var username =
             ↪  createFieldPropertyStream($('#username-id'));
```

```
112            var fullname =
                ↪  createFieldPropertyStream($('#fullname'));
113            var password =
                ↪  createFieldPropertyStream($('#password'));
114            var passwordConfirm =
                ↪  createFieldPropertyStream($('#password-confirm'));
115
116            var validUsername = username
117                .flatMap(nonEmpty)
118                .flatMap(checkValidUsernameChars)
119                .combine(
120                  username
121                      .flatMapLatest(ajaxUsernameStream)
122                      .flatMap(usernameNotTaken),
123                  function (validity, available) {
124                      if (available === true) return
                          ↪  validity;
125                      return available;
126                });
127
128            var validFullname =
                ↪  fullname.flatMap(nonEmpty).flatMap(checkCapitalizedNames);
129
130            var validPassword =
                ↪  password.flatMap(passwordLongEnough);
131
132            var passwordsEqual =
                ↪  password.combine(passwordConfirm, function
                ↪  (pass, confirm) {
133              if(pass === confirm)
134                  return pass;
135              return false;
136            }).flatMap(passwordsAreEqual);
137
138            bindFeedback(validUsername, $username);
139            bindFeedback(validFullname, $fullname);
140            bindFeedback(validPassword, $password);
141            bindFeedback(passwordsEqual, $passwordConfirm);
```

```
142
143
144             var isValid = validUsername
145                           .combine(validFullname, '.concat')
146                           .combine(validPassword, '.concat')
147                           .combine(passwordsEqual, '.concat')
148                           .map(false).mapError(true);
149             isValid.assign($('button'), "attr", "disabled");
150         }());
151     </script>
152 </body>
153 </html>
```

A.2 PHP API Resource

```
1 <?php
2 $usernames = array('bjarne', 'bja', 'b', 'chris',
   ↪ 'christian', 'c');
3 if (in_array($_GET['username'], $usernames))
4         echo "false";
5 else
6         echo "true";
7 ?>
```