

Elias Meire

Professional Bachelor Electronics – ICT

Main subject ICT

Academic year 2016/2017

Functional reactive programming in real-time web applications

Showpad
Moutstraat 62
9000 Gent
Belgium

Internship details

Intern

Elias Meire

Programme

Electronics – ICT main subject ICT

Academic year

2016/2017

Internship period

06/02/2017 - 12/05/2017

Academic promotor

Rogier van der Linde

Internship company

Showpad

Moutstraat 62

9000 Gent

Belgium

Internship supervisor

Laurens Dewaele

- **STUDENT:** Elias Meire
- **PROGRAMME:** Electronics-ICT: main subject ICT
- **BACHELOR THESIS TITLE:** Functional reactive programming in real-time web applications
- **KEYWORDS:** Functional reactive programming, real-time, web application, frontend, JavaScript, WebSocket, ReactiveX, Cycle.js
- **INTERSHIP COMPANY:** Showpad
- **ACADEMIC PROMOTER:** Rogier Van der Linde
- **INTERSHIP SUPERVISOR:** Laurens Dewaele

| |
|-------------------|
| <h2>Abstract</h2> |
|-------------------|

Web applications that update in real-time as a response to remote data changes are becoming more prominent because of new web technologies such as WebSockets and WebRTC. These applications typically depend on multiple asynchronous data sources through callback functions on events. Composing callback functions and working with asynchronous code in general can be challenging. The introduction of Promises and `async/await` to JavaScript is a big improvement in this area, but there is still a need for a more powerful abstraction. Functional reactive programming is a programming paradigm that solves this problem by modeling asynchronous code as streams. Multiple JavaScript libraries already allow developers to adopt functional reactive programming for developing web applications. This bachelor thesis explores this new paradigm and investigates how it applies to real-time web applications.

Acknowledgments

First of all I would like to thank Showpad as a whole and especially vice president engineering Chris De Mol for allowing me to do my internship in such an interesting company.

I would also like to thank everyone in the engineering team for helping me get up to speed with all of the Showpad engineering projects. In particular I would like to acknowledge Mike Warner, my team coach, for explaining me the high-level architecture of the Showpad platform and for including me in all important team activities.

Another person I would like to thank personally is Laurens Dewaele, my internship supervisor. I would like to thank Laurens for taking the time to explain the web app v2 architecture to me in detail. He really enabled me to do all the work I did on the project during my internship. The occasional pair programming sessions we did together helped me gain a deeper understanding of functional reactive programming.

I am also especially thankful to Mike, Laurens and everyone else in the share & personalize team for always taking the time to answer my questions and for listening to my input. A special thanks also goes out to Thibaut Nguyen, Arnaud Schlupp, Stijn Deschuymmer and everyone else involved with web app v2 for integrating me into the project and helping me in general.

My academic supervisor Rogier van der Linde also deserves a word of thanks for his great guidance and advice during the writing of this bachelor thesis. I also want to acknowledge him, Laurens Dewaele, Arnaud Weyts and everyone else that took the time to review this bachelor thesis and give feedback.

Finally I am sincerely grateful to my family and my girlfriend for their continuous support during the busy period writing this bachelor thesis and doing my internship.

Contents

| | |
|---|-----------|
| Abstract | 1 |
| Acknowledgments | 2 |
| Contents | 3 |
| List of Abbreviations | 6 |
| 1 Internship company | 7 |
| 1.1 Internal structure | 7 |
| 1.1.1 Engineering | 7 |
| 1.2 Culture | 9 |
| 1.2.1 Showings | 10 |
| 1.2.2 Events | 11 |
| 2 Internship assignment | 12 |
| 2.1 Web app rewrite | 12 |
| 2.2 Web app speedmonitor | 12 |
| 2.3 Showkiosk | 12 |
| 2.4 Relation to bachelor thesis | 12 |
| 3 Planning | 13 |
| 4 Research question | 15 |
| 4.1 Approach | 15 |
| 4.2 Metrics | 15 |
| 5 Real-time on the web | 16 |
| 5.1 HTTP Polling | 16 |
| 5.2 HTTP long polling | 16 |
| 5.3 WebSockets | 17 |
| 5.3.1 Libraries | 17 |
| 5.4 WebRTC | 17 |
| 6 Functional reactive programming | 19 |
| 6.1 Functional programming | 19 |
| 6.1.1 Declarative programming | 19 |
| 6.1.2 First-class functions | 20 |
| 6.1.3 Immutability | 20 |
| 6.1.4 Other functional jargon | 21 |

| | | |
|----------|--|-----------|
| 6.2 | Reactive programming | 21 |
| 6.2.1 | Operators | 21 |
| 6.2.2 | Visualizing observables | 23 |
| 6.2.3 | Debugging | 23 |
| 6.2.4 | Hot and cold observables | 24 |
| 6.2.5 | Subjects | 25 |
| 6.3 | FRP in JavaScript | 25 |
| 6.3.1 | Asynchronous Javascript | 26 |
| 6.3.2 | Current state | 28 |
| 7 | Case study: FRP in real-time data flows | 29 |
| 7.1 | User interface | 29 |
| 7.2 | Backend | 30 |
| 7.2.1 | Implementation | 30 |
| 7.2.2 | Functionality | 31 |
| 7.3 | Frontend with FRP | 31 |
| 7.3.1 | Choice of technologies | 31 |
| 7.3.2 | Architecture | 32 |
| 7.3.3 | Implementation | 33 |
| 7.4 | Frontend with imperative programming | 36 |
| 7.4.1 | Choice of technologies | 36 |
| 7.4.2 | Architecture | 36 |
| 7.4.3 | Implementation | 37 |
| 7.5 | Comparison | 38 |
| 7.5.1 | Static analysis | 38 |
| 7.5.2 | Performance | 43 |
| 7.5.3 | Readability | 46 |
| 7.5.4 | Influence of dependencies on results | 47 |
| | Conclusion | 49 |
| | References | 51 |
| | Bibliography | 54 |
| | Appendix A FRP frontend application logic | 57 |
| | Appendix B Imperative frontend application logic | 59 |
| | Appendix C Build script | 62 |
| | Appendix D Plato report for FRP implementation | 64 |
| | Appendix E Plato report for imperative programming implementation | 66 |
| | Appendix F Imperative frontend application logic (vanilla JavaScript) | 68 |

List of Abbreviations

| | |
|-------------|-----------------------------------|
| FRP | Functional reactive programming |
| QA | Quality assurance |
| Ajax | Asynchronous JavaScript and XML |
| P2P | Peer-to-peer |
| I/O | Input/output |
| API | Application programming interface |
| DOM | Document Object Model |
| JSON | JavaScript Object Notation |
| SLOC | Source lines of code |
| JS | JavaScript |

1 Internship company

I did my internship at Showpad in Ghent. Showpad is a software as a service start-up that was founded in 2011 by ex-employees of In The Pocket, a mobile agency. Since then the company has grown at a fast pace, today Showpad has over 1000 customers and over 200 employees in offices in Ghent, London, San Francisco and Portland [1].

Showpad provides a business-to-business service that aims to align marketing and sales departments by making marketing materials easily accessible and shareable. Showpad achieves this by hosting a platform and developing native applications for Android and iOS as well as a web application.

Showpad aims its service at enterprise customers with over 250 employees. Showpad's customers include Coca-cola, Audi, Johnson & Johnson and many other big names [2]. These customers pay Showpad a monthly subscription fee per user for the software as a service.

Showpad has been doubling its revenue every year for the past 4 years. At the end of 2015 Showpad's annual recurring revenue exceeded \$10 million [3]. In May 2016 Showpad closed a \$50 million funding round led by Insight Venture Partners [1].

1.1 Internal structure

Internally Showpad is divided into 5 departments, listed in order of descending size these are: customer success, engineering, sales, marketing and employee success. As can be deduced from this order, Showpad invests heavily in customer experience. Showpad prides itself on its very short customer issue cycle time and a strong synergy between customer success and engineering is essential to achieve this.

1.1.1 Engineering

Teams

The engineering department is divided into 6 smaller teams with about 10 members. These teams are not divided by product or discipline but rather by the functionality they implement and maintain. An example of this is the create & present team. That team maintains all user functionality related to creating and presenting content in Showpad.

The engineering teams in Showpad are discover & measure, distribute & collaborate, share & personalize and create & present. Every team consists of frontend and backend developers as well at least one quality assurance engineer and a team coach. There are also two teams that aren't linked to specific functionality. Those teams are the mobile & learn team and the architecture team because these are two

smaller teams that can't be linked to a specific user functionality. Even though these are separate teams there is still a lot of collaboration between different teams.

During my internship I was a part of the share & personalize team. This team maintains all features related to sharing and personalizing content in all Showpad products (except the mobile apps).

Guilds

In Showpad engineering there are guilds for every development discipline. These 4 guilds are the frontend guild, the backend guild, the architecture guild and the QA guild. Every guild has a separate group chat to share interesting articles or new developments in their areas of expertise. The guilds also have a weekly meeting to talk about any topic within their field. In these meetings every guild member has the opportunity to give a presentation or workshop on something they find interesting in their discipline.

During my internship I was a part of the frontend guild. The last week of my internship I gave a presentation during the guild. The presentation was about common pitfalls in functional reactive programming and included small code recipes and demos [4].

Kanban

Every team has a Kanban board [5] on which they receive and create new development tickets. These tickets are then taken through a development cycle which includes a peer code review and a testing session (see figure 1.1). Sometimes there are separate Kanban boards for epics. Epics are large features that typically require a long time in development and are not specific to one team. An example of an epic I took part in during my internship is the complete rewrite of the web application in Angular 2.

For distributing tickets between developers Showpad engineering uses a pull system instead of a push system. This means that developers self-assign tasks they feel comfortable working on instead of tasks being pushed to developers by a team coach.

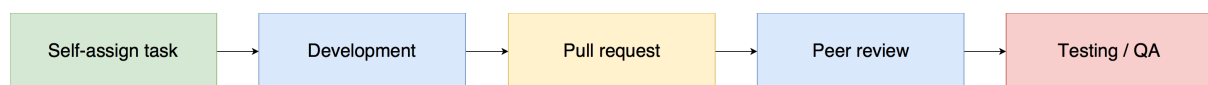


Figure 1.1: Ticket development cycle in Showpad

Continuous integration

Showpad engineering puts a lot of effort into continuous integration and deployment. The development process is largely automated (see figure 1.2). Every commit that is pushed to a git repository gets built and tested automatically by a build server. Furthermore every pull request that is approved in a code review gets deployed to a minion where it gets tested by a QA engineer. If the QA engineer approves the changes and verifies the functionality the pull request will be deployed on the staging servers. On the staging servers a last sanity check is done to ensure everything is working. If there are no issues

on staging the code will be added to a release and deployed to production in cycles of about a week on average. This very fast development cycle results in a rapidly improving product and a very quick response time to customer reported issues.

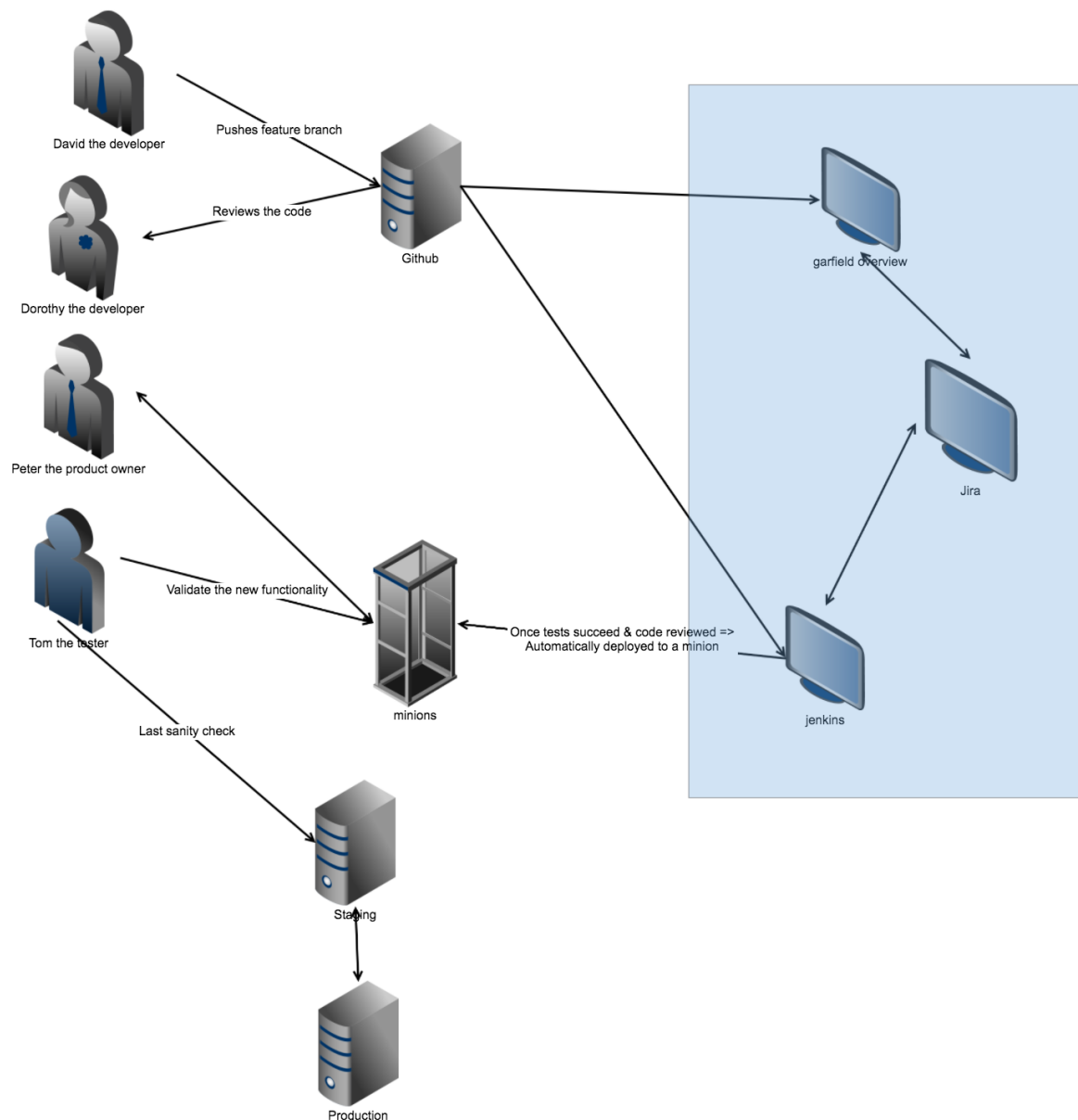


Figure 1.2: *Showpad developer workflow diagram*

1.2 Culture

Showpad has a very open culture that is typical for a start-up in the technology scene. The office is designed to be as open as possible to encourage interdepartmental collaboration and create a good atmosphere in general (see figure 1.3).



Figure 1.3: *Showpad office atmosphere*

1.2.1 Showings

Showpad outlines 6 characteristics called showings that a good Showpad employee should have. These showings are shown in figure 1.4.







- 
KEEP IT SIMPLE
 - We understand that less is more
 - We aim for **maximum impact**
- 
TRANSPARENCY BY DEFAULT
 - **We share** fast
 - We share the why
- 
GIVE TO GROW
 - We share knowledge for mutual growth
 - We believe **feedback** is a gift
- 
BE HUMBLE
 - We take our job seriously but ourselves not so much
 - We value **team success** over personal wins
- 
TAKE OWNERSHIP
 - We think and act like leaders
 - We are **pro-active**
- 
BE PASSIONATE
 - We love **what we do**
 - We do what we love

Figure 1.4: *The 6 showings*

1.2.2 Events

Showpad organizes a lot of events for its employees and customers. Every Friday evening there is an event for all employees where they can have drinks together to set off the weekend. Every other week there is also a Showpie on Friday. This is an event where all employees in all offices get together in a video call and there is a moment to ask questions to the leadership. Sometimes Showpie also includes a speech or presentation by the CEO.

Showpad engineering organizes their own yearly developer conference in Ghent called GEARS [6]. This is a free event open to all web developers. The event hosts talks and presentations by Showpad engineers and external speakers from leading technology companies. There are open donations on the event that go to a charity of Showpad's choosing.

Lastly Showpad organizes a yearly event for its customers called Showtime. Showtime gives Showpad customers a chance to share their experience with the product [7].

2 Internship assignment

My internship assignment at Showpad will consist of 3 main components: the web app rewrite, the web app speed monitor and the Showkiosk.

2.1 Web app rewrite

The largest part of my internship at Showpad will be spent doing work on the web app v2. Showpad web app v2 is a complete rewrite of the original web app in Angular 2+ and RxJS. The decision to rewrite the web app was made in July 2016 because of two major reasons. The first reason was that web app v1 was written as one big monolith. As a result development on the project was tedious, especially for new developers. The second reason is that the technologies that v1 was made with have become heavily outdated (ex. Angular 1).

During my internship I will work on the web app v2 until it gets released to customers. Afterwards I will still be part of development team on web app v2. This development will include fixing bugs and implementing new features.

2.2 Web app speedmonitor

The web app speed monitor will be a data visualisation that will measure and display the performance of the web app in real-time. Some of the technologies used in this project will be React, RxJS, Redux and Websockets. This application will be developed from scratch during my internship.

2.3 Showkiosk

The Showkiosk is an internal tool to display important message on screens throughout the company. Key technologies used in this project are Angular 2+ and Google Firebase.

2.4 Relation to bachelor thesis

The assignments in my internship are all related to the topic of my bachelor thesis. These projects will not be covered in detail in this thesis however, but lessons learned from using the technologies on these projects will be used as a basis to understand them and write this bachelor thesis.

3 Planning

| Step | Content | Target date | Real date |
|-------------------|--|-------------|-----------|
| Deadlines | | | |
| 1. | Create planning | | |
| 2. | Decide title for bachelor thesis | 20/03 | 16/03 |
| 3. | Submit first part of bachelor thesis | 27/03 | 27/03 |
| 4. | Intermediate evaluation by internship mentor | 31/03 | 28/03 |
| 5. | Submit second part of bachelor thesis | 24/04 | 24/04 |
| 6. | Submit bachelor thesis | 26/05 | 26/05 |
| Research | | | |
| 1. | Functional Reactive programming | 20/02 | 19/02 |
| 1. | Declarative vs. imperative programming | 20/02 | 19/02 |
| 2. | Real time on the web (Websockets, long polling...) | 27/02 | 28/02 |
| 2. | Reactive implementations in Javascript | 27/02 | 26/02 |
| 3. | Elm: A purely functional language for the web | 06/03 | 10/03 |
| 4. | FRP in user interfaces | 13/03 | 13/03 |
| 4. | Further research on FRP | 13/03 | 13/03 |
| 5. | Matching FRP with Real-time applications | 20/03 | 23/03 |
| Internship | | | |
| 1. | <u>Web app v2 (Angular 2+, RxJS rewrite)</u> | | |
| | Verify existence of issues reported in v1 that are still in v2 and help solving them | 20/03 | 17/03 |

| | | | |
|----|--|-------|----------|
| | Maintenance, bug fixes and improvements | 12/05 | 05/05 |
| | Help develop new feature (Locked pages project) | 12/05 | Canceled |
| 2. | <u>Speed monitor (Websockets, React)</u> | | |
| | Help develop team tool to monitor speed of web app | 12/05 | 12/05 |
| 3. | <u>Showkiosk (Angular 2+, Google Firebase)</u> | | |
| | Help develop Internal communication tool to display messages throughout the entire company (1day/week) | 12/05 | 12/05 |

Table 3.1: *Planning for internship and bachelor thesis*

4 Research question

In this bachelor thesis research is conducted on how functional reactive programming can be beneficial to development of real-time web applications. The research question that is answered is the following: "What advantages and disadvantages does functional reactive programming have when developing real-time web applications?".

4.1 Approach

The first part of this bachelor thesis researches FRP and real-time dataflow on the web today. Because this bachelor thesis is applied to the web all code examples are written in JavaScript.

To come to a conclusion a practical case study is made. The research question is evaluated by implementing a simple case in a traditional imperative programming style and in a declarative FRP style. The metrics defined in section 4.2 will then be used to compare the two implementations and formulate a conclusion.

The case that is implemented in this bachelor thesis is a simple text-based chat application. This is the simplest example of a real-time web application with full-duplex communication.

4.2 Metrics

First static analysis is performed on the source code of both implementations. Metrics that result from this analysis such as bundle size and complexity metrics are then used to compare the implementations. After that the runtime performance of both implementations is compared in two scenarios. Finally there is a comparison based on the readability of the source code.

5 Real-time on the web

5.1 HTTP Polling

The simplest way to do real-time on the web is to poll a server via Ajax at a certain interval. This is a simple but very primitive way to update data in real-time. This technique has multiple downsides. Firstly the communication is not really real-time and your data is always out of date because of the usage of an interval. Secondly this technique increases load on your server because it has to handle a lot of requests from the clients. This makes scaling with an HTTP polling architecture very cumbersome [8]. Lastly this technique cannot do two-way communication between server and client so it is not suitable for real-time communication [8].

5.2 HTTP long polling

HTTP long polling is a variation on HTTP polling. The client still polls the server for new data but the server keeps the request open until it has new data. After the client gets a response it immediately sends a new request, repeating the process [9] (see figure 5.1). This technique resolves most downsides of classic HTTP polling but it still cannot facilitate two-way communication between server and client [8].

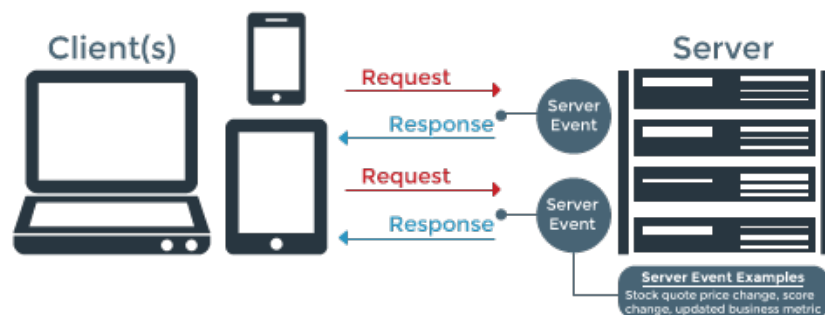


Figure 5.1: Long polling [9]

5.3 WebSockets

HTML5 WebSockets are a full-duplex communication channel (see figure 5.2) that operates through a single socket over the web. HTML5 WebSockets are not just another small improvement over conventional HTTP communications [8], it is a completely new communication protocol. Under the hood WebSockets are built on TCP; they do not use HTTP except for establishing the connection [10]. Browser support for WebSockets is excellent, all major browsers support them [11].

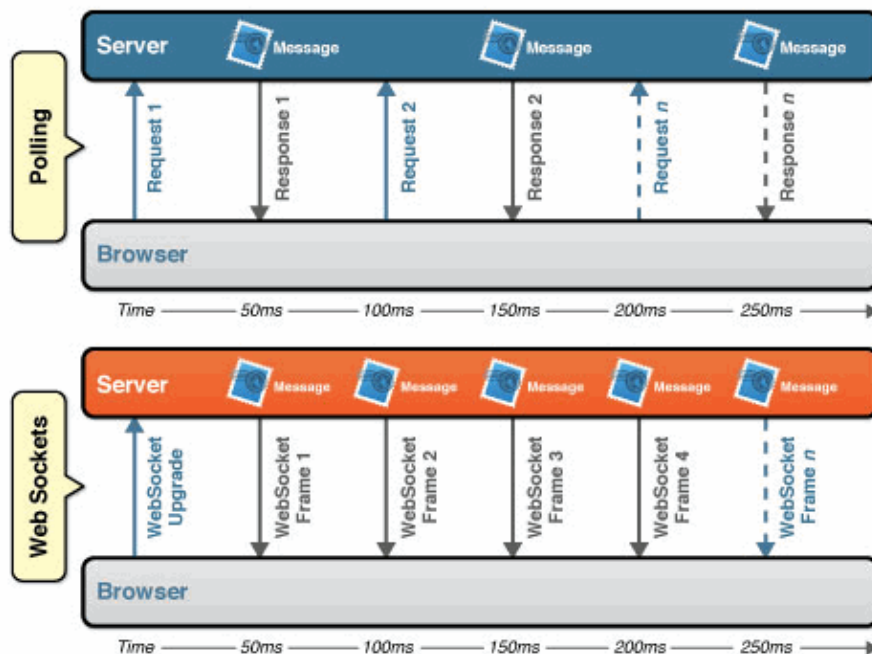


Figure 5.2: Comparison of HTTP polling and WebSockets [8]

5.3.1 Libraries

There are many libraries that abstract away some of the complexity from the WebSocket protocol. The most popular WebSocket libraries at the time of writing are Socket.IO, Primus, ws and Faye. The most popular reasons to choose a library over using the vanilla version are elegant fallbacks for browsers that do not support WebSockets and automatic reconnection.

5.4 WebRTC

WebRTC is a full-duplex communication protocol that was developed and open sourced by Google in 2010 [12]. After that WebRTC was standardised by the IETF and the W3C [12]. Now there are implemented open standards for real-time, plugin-free video, audio and data communication. WebRTC allows real-time peer-to-peer communication of audio, video and arbitrary data between browsers. A

server is only needed to set up the initial connection between peers, this process is called signaling [13] (see figure 5.3).

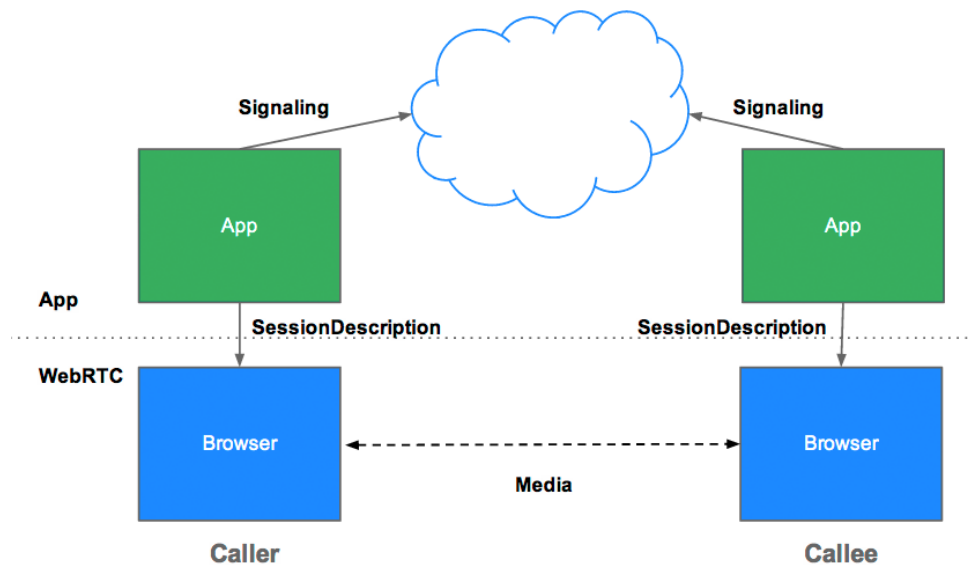


Figure 5.3: *Signaling process for WebRTC [12]*

WebRTC's P2P connection model further increases scalability and simplicity of real-time applications [13]. However WebRTC also has some caveats. Even though it has been around for 4 years now browser support is still lacking at the time of writing [14]. It is also hard to implement WebRTC in a client-server architecture [15] as it was primarily made for P2P communication.

6 Functional reactive programming

Functional reactive programming is a programming paradigm that integrates the idea of asynchronous dataflows into functional programming. This provides an elegant way to express computation for areas that involve a lot of time related variables such as animation and user interface rendering [16][17].

6.1 Functional programming

Functional programming is a programming paradigm that models computation as a composition of pure functions without mutating any state [18]. It is a declarative programming paradigm. One of the biggest advantages of functional programming is that by using pure functions it becomes much easier to reason about the behavior of a program [18].

Functional programming has its origins in lambda calculus, a system in mathematical logic for expressing computation based on operations and variables [19].

6.1.1 Declarative programming

In declarative programming a program describes what to do, rather than how to do it. This is the opposite of traditional imperative programming [20]. The difference between the two is best explained by example. Listing 6.1 and listing 6.2 both show an implementation of a function called `double()`. The function takes an array and returns another array with all elements doubled. Listing 6.1 shows an imperative implementation of this function, the function explicitly says to loop over the elements, double them and add them to an empty array. Listing 6.2 shows a declarative implementation of this function, it only says what we want to happen to the elements in the array.

Listing 6.1: *Imperative implementation of the double function*

```
function double (arr) {  
  let results = [];  
  for (let i = 0; i < arr.length; i++) {  
    results.push(arr[i] * 2);  
  }  
  return results;  
}
```

Listing 6.2: *Declarative implementation of the double function*

```
function double (arr) {  
  return arr.map(item => item * 2);  
}
```

6.1.2 First-class functions

In functional programming functions are first-class citizens. This means that functions can be treated like any other value. They can be created, stored in data structures and passed into or returned from other functions.

Pure functions

Pure functions have two defining characteristics. The first is that the function's result only depends on its arguments, given the same arguments the function always returns the same value [20]. The second characteristic is that pure functions do not cause any side effects such as mutations to data structures or output to I/O devices [20].

Higher order functions

Higher order functions are functions that either take a function as an argument, return a function or both. This is a simple concept that makes function composition possible. Function composition helps to keep functional programs readable and easy to reason about [20].

An example of a higher order function in JavaScript is the `map()` function of the Array prototype (see listing 6.2). This function takes another function as its argument. The `invokeTwice()` function in listing 6.3 is an example of a higher order function that returns a function.

Listing 6.3: *Higher order function composition*

```
const invokeTwice = f => subject => f(f(subject));
const double = x => x * 2;
const quad = invokeTwice(double);

quad(10); // 40
invokeTwice(double)(10); // 40
```

The `invokeTwice()` function takes a function as an argument and returns a function that applies the first function twice on its argument. The example also shows how `invokeTwice()` can be composed together with other functions such as `double`. A function like `invokeTwice()` that takes one argument at a time and returns a function taking the next argument is called a curried function. [18].

6.1.3 Immutability

Functional programming promotes the use of immutable data structures, most functional languages even enforce it [18]. Immutability means that a data structure cannot be changed. If an operation is done on an immutable object, it copies the value, mutates it and returns it as a new object. This helps eliminate bugs where an object enters a state that was not predicted by the programmer [20].

6.1.4 Other functional jargon

There are many other terms associated with functional programming but these are out of scope for this thesis. This chapter only serves to give a basic understanding of the functional programming paradigm.

6.2 Reactive programming

Reactive programming is a programming paradigm that deals with asynchronous data streams [21]. These data streams then propagate their changes to other parts of the application; this is the observer pattern [22]. Reactive programming has three main objects: an observable or a stream, an observer or a subscriber and a scheduler [21].

An observable emits values over time. An observable can be infinite or it can complete after a certain number of emitted values. Observers can subscribe to these streams and get notified whenever the observable emits a value, throws an error or completes (see listing 6.4 for an example). Schedulers decide on which thread an observer's code should run and when.

Listing 6.4: *An observable and observer in RxJS*

```
const someObservable$ = Observable.from([1,2,3]);

someObservable$
  .subscribe(
    value => console.log(value),
    error => console.err(error),
    completed => console.log('completed')
  );

// output: 1, 2, 3, completed
```

6.2.1 Operators

Typically an FRP library or language includes a lot of operators [21] so that observables can be easily manipulated. In FRP operators are pure functions that take an observable and return another observable. A simple example of an operator is the delay operator. This operator is available in RxJS and returns the input observable with every value delayed by a specified amount of time [23] (see figure 6.1). Because operators return a new observable, operators can be easily chained after one another.

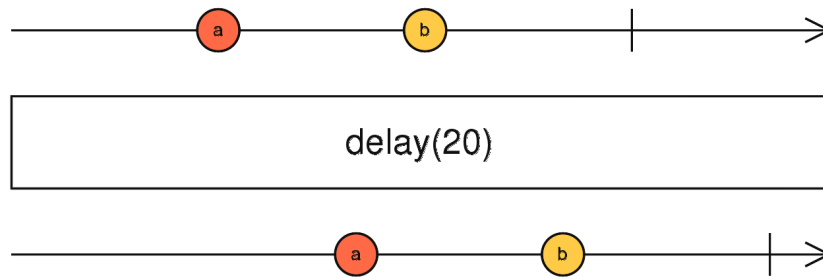


Figure 6.1: *The delay operator in RxJS: marble diagram [23]*

An example of operators on a data stream can be seen in figure 6.2, this example illustrates a stream of clicks by a user. The events in the stream are first grouped by throttling them for 250ms. Afterwards they are mapped to the length of the groups and finally they are filtered so that only groups of clicks equal or larger than 2 remain. This example shows really well how reactive programming helps with handling complex operations on events in modern applications.

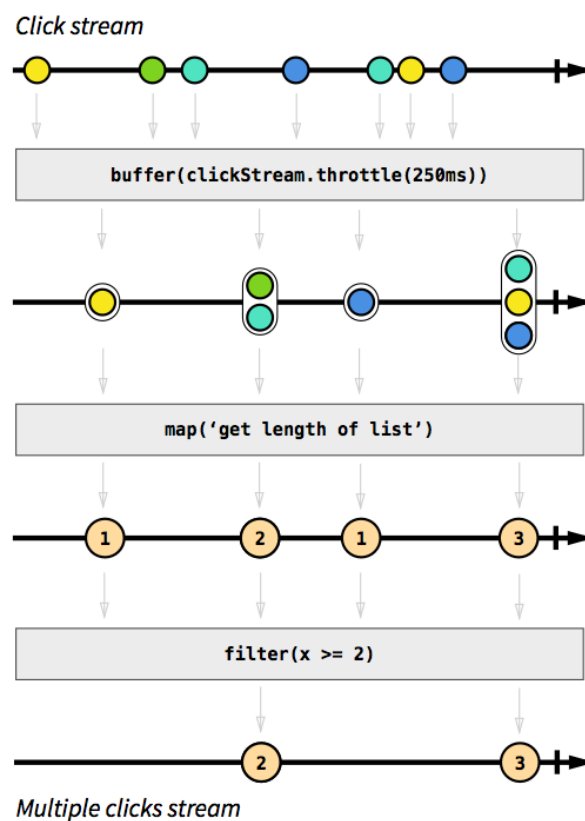


Figure 6.2: *Stream of clicks [21]*

6.2.2 Visualizing observables

The biggest roadblock in adoption of reactive programming by developers has been the steep learning curve. Reactive programming forces the developer to think about dataflow in a different way. In reactive programming data arrives over time and is immutable. Essential in learning to think reactive is being able to visualize how observables work over time.

Marble diagrams are the main tool that is used today to visualize observables. These diagrams can be great tools to not only understand the core concepts of reactive programming but also understand the functionality of individual operators (see figure 6.1 and 6.2).

6.2.3 Debugging

The main area where reactive programming falls short today is debugging. Since operators tend to be one line arrow functions it is not possible to set a breakpoint inside them. This results in a lot of developers either using console statements or having to convert operator functions to functions with a body every time they want to debug. Moreover stack traces tend to be longer and less useful than those from imperative programming. Because everything is asynchronous and a lot of work is done internally in the FRP library, stack traces tend to show only the errors internal to the library which is not useful for the developer [24]. An important side note is that Paul Irish announced new features coming to Chrome Devtools during Google I/O 2017 that will allow developers to debug asynchronous code more efficiently [25]. Stack traces will become more descriptive and developers will be able to step inside one line arrow functions such as `Promise.then()` [25].

RxJS 5 is a complete rewrite of RxJS 4 and one of its biggest improvements is its shorter stack traces making it easier for developers to debug their code [24]. Other tools one can use to debug and reason about reactive code are marble diagrams and dependency graphs [24]. Marble diagrams let developers visualize observables and find mistakes in their reasoning. Dependency graphs help the developer understand what observables an observable depends on and for what. This helps identify problems where an observable the current observable depends on is not behaving the way the developer expects it to. An example of a dependency graph is shown in figure 6.3.

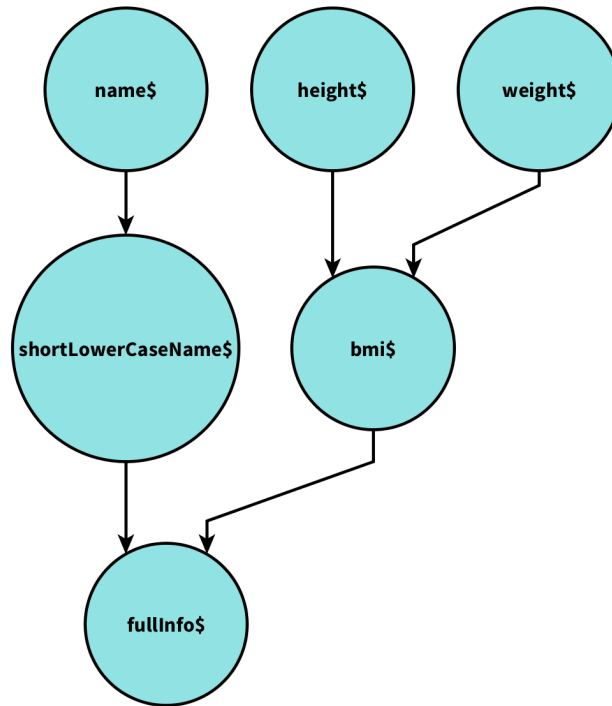


Figure 6.3: A dependency graph of observables in a BMI calculator [24]

Figure 6.3 above shows the dependency graph of a simple Body Mass Index (BMI) calculator. The code associated with this example can be seen in listing 6.5. The code example assumes that the input streams `name$`, `weight$` and `height$` are already instantiated before this code runs.

Listing 6.5: Reactive BMI calculator source code [24]

```
const shortLowerCaseName$ = name$
  .map(name => name.toLowerCase())
  .filter(name => name.length < 5);

const bmi$ = weight$
  .combineLatest(
    height$,
    (weight, height) => Math.round(weight / (height * height * 0.0001))
  );

const fullInfo$ = shortLowerCaseName$.combineLatest(bmi$);
```

6.2.4 Hot and cold observables

Observables can be either hot or cold. The way the producer is linked with the observable defines whether it is hot or cold. A producer is an object that provides values to an observable. The producer can be an iterator, an event, a WebSocket etc. The pseudocode in listing 6.6 shows how a producer is

linked up to an observable for hot and cold observables.

Listing 6.6: *Hot and cold observables in RxJS [26]*

```
// Cold observable
const cold = new Observable((observer) => {
  const producer = new Producer();
  // have observer listen to producer here
});

// Hot observable
const producer = new Producer();
const hot = new Observable((observer) => {
  // have observer listen to producer here
});
```

When creating a cold observable a producer is made for each observer. This means that if there is no observer subscribed there is also no producer being instantiated. This is the reason why observables in RxJS need to be subscribed to in order to emit anything at all. Cold observables are unicast [26]. This means that every producer only emits its data to one observer. As a result this can cause issues when working with very rapidly changing data such as time. Because every observer has its own producer, the time at which observers receive the same data can differ slightly [26].

For a hot observable the producer is set up even if no observer is subscribed to it. When multiple observers subscribe to the same observable the producer is shared. Emitted data from the producer is multicasted to all observers [26]. In most FRP libraries including RxJS, observables are cold by default. Observables in RxJS can be made hot with the `publish` and `share` operators [26], subjects (see section 6.2.5) are always hot.

6.2.5 Subjects

Subjects are an additional data structure provided in some FRP libraries. Subjects behave exactly like observables but their data does not come from a producer. Instead data can be pushed manually to the subject [27]. This helps bridge the gap between imperative programming and reactive programming. Subjects make it easy for a programmer to integrate stateful data into an observable. Since subjects are compatible with all observable operators they can be combined with other observables. It is considered to be a bad practice though and whenever it can be avoided it is advised to use observables over subjects [27].

6.3 FRP in JavaScript

Because of JavaScript's unopinionated nature, choice of programming paradigm is left to the developer. JavaScript also has a lot of the ingredients needed for FRP built into the language. Functions in JavaScript are already first-class citizens of the language [18]. Arrow functions allow JavaScript functions to be

Listing 6.7: *Fetching data with nested callbacks*

```
getData(function (err, x) {
  if(err) {
    console.error('Oh no! an error: ${err}');
    return;
  }
  getMoreData(x, function (err, y) {
    if(err) {
      console.error('Oh no! an error: ${err}');
      return;
    }
    other(x, y, function (err, z) {
      if(err) {
        console.error('Oh no! an error: ${err}');
        return;
      }
      ...
    });
  });
});
```

reasoned about as lambda calculus and finally ES2015 introduced a few functional array functions such as map, reduce and filter [28].

Because of these characteristics JavaScript is an excellent language for functional reactive programming.

6.3.1 Asynchronous Javascript

Dealing with asynchronous code has always been one of the biggest challenges of programming for the web. Over the years developers have employed different techniques to handle asynchronous code in JavaScript.

Callbacks

Callbacks are the simplest way to handle asynchronous code in JavaScript. The principle is simple; a callback function is passed as an argument with another function. That function will then call the callback function when it finishes its work.

The problem with callbacks is that they quickly become unreadable when the programmer needs to compose different asynchronous actions. Listing 6.7 is an example of 3 asynchronous functions running one after the other. This example code is already pretty complex even though it accomplishes a very simple task.

Promises

Promises are an alternative way to handle asynchronous code. Promises were standardized in ES2015 but have been around for years in various libraries such as Bluebird [29]. The Promise API allows the developer to compose asynchronous code in a more declarative manner. A promise can either resolve and deliver a value or reject and throw an error.

Listing 6.8 shows the example of listing 6.7 rewritten with the JavaScript Promise API. The `Promise.catch()` function handles all errors on a chain of promises.

Listing 6.8: *Fetching data with the Promise API*

```
getDataPromise
  .then(x => getMoreDataPromise(x))
  .then(y => otherPromise(y))
  .then(z => ...)
  .catch(err => {
    console.error('Oh no! an error: ${err}');
  });
```

If the promises in the example would not depend on the result of the previous one the `Promise.all()` function could be used. This function completes when all promises have resolved. The `Promise.all()` function throws an error if one of the promises rejects [30].

Async/await

Async functions and the `async/await` syntax was recently introduced to JavaScript in ES2017 [31]. It is a new language construct to handle asynchronous code. The `await` keyword blocks code execution until an async function has completed. The example used in listing 6.7 and listing 6.8 is rewritten below in listing 6.9 using the `async/await` syntax.

Listing 6.9: *Fetching data with the `async/await` keywords*

```
async function getAllData() {
  try {
    const x = await getData();
    const y = await getMoreData(x);
    const z = await other(y);
    ...
  } catch(err) {
    console.error('Oh no! an error: ${err}');
  }
}
```

Functional Reactive Programming

FRP provides a more versatile API to handle asynchronous code than Promises and `async/await`. As Akash Agrawal correctly notes in his article "What Promises Do That Observables Can't" observables completely overshadow promises in terms of functionality [32].

Listing 6.10 shows the example from the listings in the previous paragraphs rewritten using RxJS observables. The `getMoreDataObservable()` and `getOtherObservable()` functions in this example return an observable based on their parameter. While the implementations with promises and `async/await` are slightly more concise, FRP offers a lot more functionality than both. RxJS observables for example provide operators to cancel, buffer and debounce asynchronous code [33]. Promises or `async/await` provide no API for those functionalities.

Listing 6.10: *Fetching data with FRP (RxJS)*

```
dataObservable$
  .flatMap(x => getMoreDataObservable(x))
  .flatMap(y => getOtherObservable(x))
  .catch(err => {
    console.error('Oh no! an error: ${err}');
  })
  .subscribe(z => {
    ...
  });
```

6.3.2 Current state

Many libraries already bring FRP to JavaScript. The most popular ones at the time of writing are RxJS, xstream and Bacon.js [21]. There is also a proposal out there to bake an observable data structure into a future version of ECMAScript [34].

In 2013 Evan Czaplicki and Stephen Chong published a thesis at the Harvard University about the creation of Elm [35]. Elm is a practical FRP language with two important features: high-level abstractions to support simple, declarative FRP and purely functional graphical layout calculation [35]. Elm is also known for its excellent tooling and the absence of runtime errors in the language. Most importantly the Elm language is built on top of JavaScript and is made with the web in mind.

The FRP paradigm has been around for a very long time but it is still very new to the web platform. The need for a good abstraction to handle asynchronous code can be a driving factor to increase adoption of FRP in JavaScript. Possible support for observables in the JavaScript language can also further increase developer adoption in the future.

7 Case study: FRP in real-time data flows

In this chapter a real-time chat application is developed and analyzed. The chat application is implemented twice in different programming styles. The first implementation uses FRP for all application logic and Cycle.js to render to the Document Object Model. The second implementation uses imperative programming for all application logic and JQuery for DOM manipulation.

The backend implementation is the same for both frontend implementations. Real-time communication between server and client is implemented using WebSockets. WebSockets was chosen over HTTP long polling because long polling does not implement a full-duplex communication channel. WebSockets are chosen over WebRTC data channels because of browser support and maturity of the protocol. Furthermore WebRTC is primarily meant for P2P applications and is cumbersome to implement in a client-server architecture [15].

The source code of this application is open source and is available on GitHub [36]. The complete frontend application logic from both applications can be found in appendix A and B. Excerpts from the source code are included throughout this chapter to explain implementation details.

7.1 User interface

The application features a very simple user interface (see figure 7.1). Users can enter their name and a message in the text inputs in the bottom bar. They can then submit the form to send a new message over the WebSocket. The server will then broadcast this message to all clients connected to the WebSocket.

On the frontend the messages are displayed with an avatar next to them that is automatically generated from the username of the sender. The side the messages are rendered on is dependent on whether the current user is equal to the sender of the message. A user's own messages appear on the right and messages from other people appear on the left.

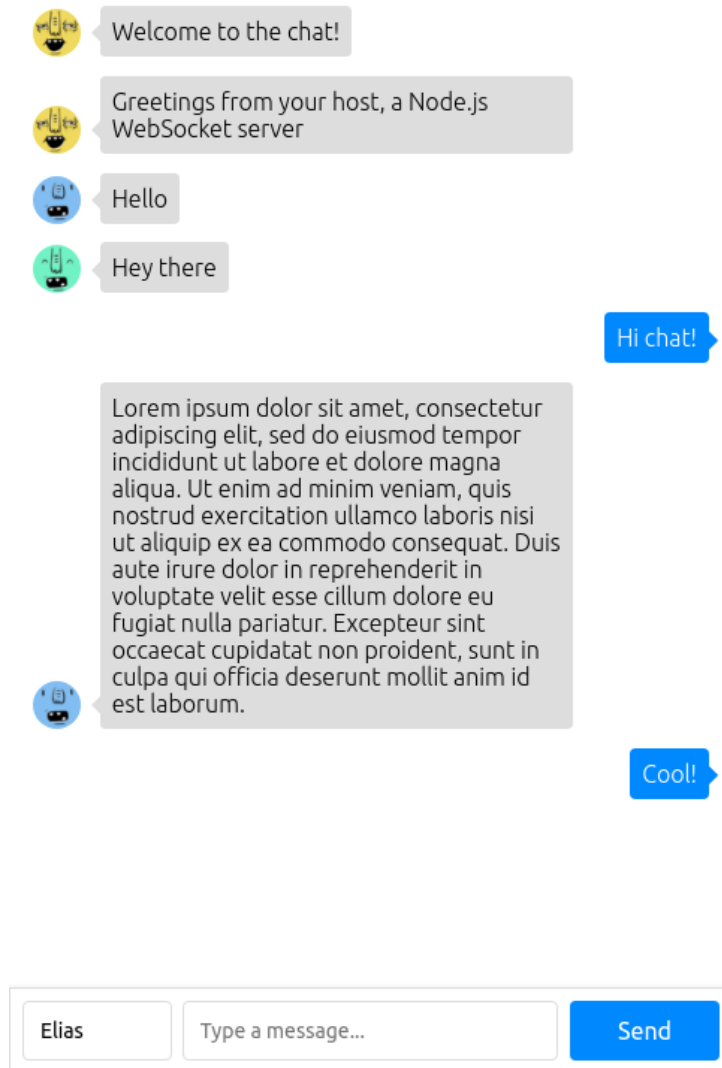


Figure 7.1: *The chat application interface*

7.2 Backend

7.2.1 Implementation

The backend is implemented in Node.js. It makes use of the Express framework to set up an HTTP server and host the static files from the frontend. The WebSocket implementation that is used is `µWebSockets`. `µWebSockets` is a lightweight WebSocket implementation written in C++ with simplicity, performance and scalability as its main objectives [37].

`µWebSockets` was chosen because of its simplicity and performance. This application is meant to be a very minimal implementation of a chat application and as a result it does not need any extra features on top of the WebSocket protocol. `Socket.IO` or other popular WebSocket libraries would have been overkill for this application as it does not need any of the extra features it provides.

Messages are not persisted on the server because it would add extra dependencies and potential performance bottlenecks. This case study is focused on the comparison of FRP and imperative programming for the web. As a result it was attempted to add as little dependencies as possible to focus on that comparison.

7.2.2 Functionality

The backend sets up the WebSocket using μ WebSockets and listens for connections. When a client connects to the WebSocket the server sends out a welcome message. When a client sends a message over the WebSocket the server broadcasts this message to all clients that are currently connected.

7.3 Frontend with FRP

7.3.1 Choice of technologies

FRP library

RxJS is used as the FRP library for this application. This choice is made mainly because of the popularity of the library. ReactiveX is a very mature cross-language FRP library with big companies like Microsoft and Netflix backing it and using it in their own applications [38]. The JavaScript version of ReactiveX (RxJS) is currently the most popular FRP library for JavaScript by a large margin. The npm installation statistics support this statement [39][40][41].

Xstream and Most.js are also considered. Both these FRP libraries feature better performance than RxJS and are only a fraction of the size [39][40][41]. But ultimately the performance and bundle size of this application is not the priority. Using RxJS has the advantage of more people being familiar with its concepts and operators.

DOM abstraction

For a DOM abstraction various frameworks and libraries are considered. Since the objective is to use FRP in the application, frameworks that play nice with FRP have the advantage. Angular is the first framework that comes to mind in this category. Angular has built-in support for observables and even uses RxJS internally for its core APIs. But because of the size and complexity of Angular there is too much overhead for this application. This application is meant to be as minimal as possible and Angular does not fit the bill in that department.

Cycle.js is the next framework to be considered. Cycle.js is a functional and reactive JavaScript framework for predictable code [42]. It is written by André Staltz who is also a core contributor of RxJS and the author of xstream [43][41]. Cycle.js features a very minimal API and allows the programmer to see the application as a pure function. To use Cycle.js the programmer has to only import one function from the framework: `run()`. This makes code easy to understand for newcomers since there are no new

framework APIs to learn. Cycle.js is also FRP library agnostic and thus allows the programmer to use their library of choice [42].

Because of its minimal API and very functional architecture Cycle.js was chosen as the DOM abstraction for this application. The architecture used in Cycle.js is further explained in section 7.3.2.

7.3.2 Architecture

Cycle.js lets the developer see the app as a pure function. This `main()` function takes source streams as an argument and returns sink streams [42]. Figure 7.2 shows a diagram of how this architecture works in practice.

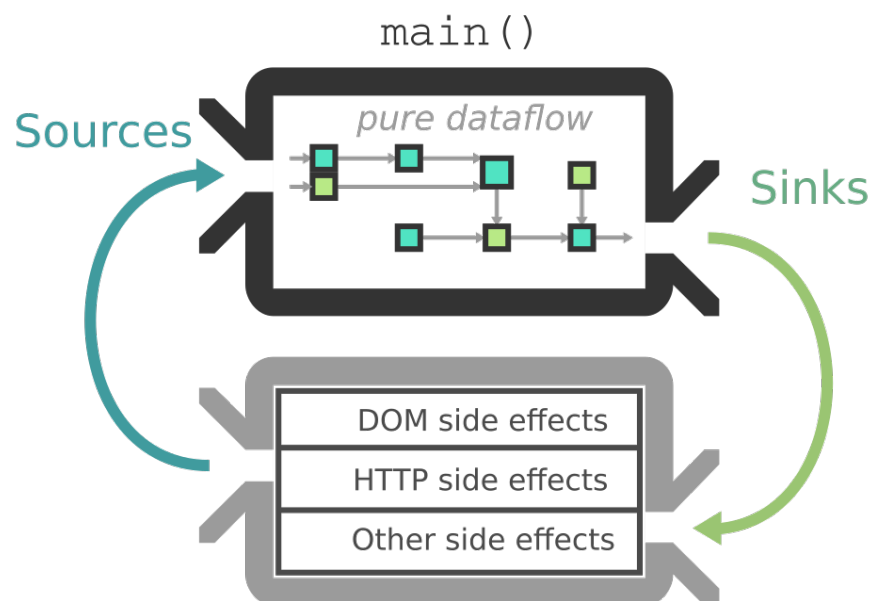


Figure 7.2: *Cycle.js application architecture [42]*

The bottom function shown in figure 7.2 is called a driver. Drivers handle side effects towards various outputs such as the DOM. Drivers take sinks as an argument and return sources for the `main()` function to use [44]. Driver functions can be seen as the inverse of the `main()` function. Cycle.js provides a few basic drivers for the DOM and for HTTP requests. Cycle.js is very extensible [44] and as a result other drivers written by the community can be found on open source platforms.

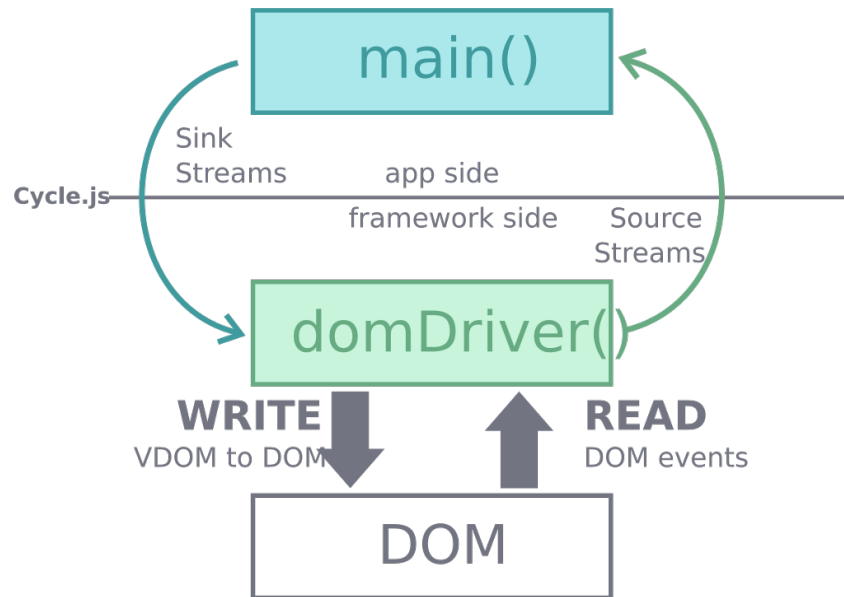


Figure 7.3: *Cycle.js DOM driver diagram [44]*

The diagram shown in figure 7.3 is an example of a Cycle.js driver. It represents the built-in DOM driver. This driver handles writing DOM changes from the sink stream and provides DOM elements and events in the source stream.

Custom WebSocket driver

For this project a custom driver was developed to set the WebSocket up as a source and a sink stream. The driver is available as a package on npm [45] so that it can be used by other users of Cycle.js. The source code is open source and is available on GitHub [46].

The driver is a wrapper around the `WebSocketSubject` object provided by RxJS [47]. Internally the driver acts as an adapter that forwards the messages from the sink stream to the `WebSocketSubject`. The driver returns the `WebSocketSubject` as a whole as the source stream, that way the developer has full control over it in the `main()` function. Special attention was given to ensure that the driver is also compatible with other FRP libraries such as xstream and Most.js.

7.3.3 Implementation

When developing an application with the concept of source and sink streams the developer first defines the input streams needed for the application. These input streams can be user input, a real-time database etc.

In the case of this application these input streams are inputs from the user through DOM events and real-time messages from the WebSocket. In the source code the input streams are defined as in listing 7.1. The input streams are derived from the sources given as an argument to the Cycle.js `main()` function. In some cases the input streams might even be just a source, without any mutation.

The `scan()` operator on the WebSocket source collects all messages received up until this point in an array. The marble diagram in the comment on line 2 shows how this works in practice. `{m}` represents a single message object.

Listing 7.1: Definition and instantiation of the input streams

```
const messages$ = ws.scan((acc, m) => [...acc, m], []);
// --[{m}]{m}--[{m}, {m}]{m}--[{m}, {m}, {m}]{m}-->
const formSubmit$ = DOM.select('#form').events('submit');
const senderInput$ = DOM.select('#sender').events('input');
const messageInput$ = DOM.select('#message').events('input');
```

After the input streams are defined the developer can think about what sinks (output streams) the application will output towards. In the case of web applications one sink that is always used is the DOM. `LocalStorage` is an example of another output stream that some applications use [48]. In the case of this application the used sinks are the DOM and the WebSocket.

Now that the sources and sinks are known all that is left to do is to combine and manipulate the input streams until they are in the required format for the sinks. Before writing any code it can be useful for the developer to make a dependency graph to see which output streams depend on which input streams. The dependency graph for this chat application is shown in figure 7.4.

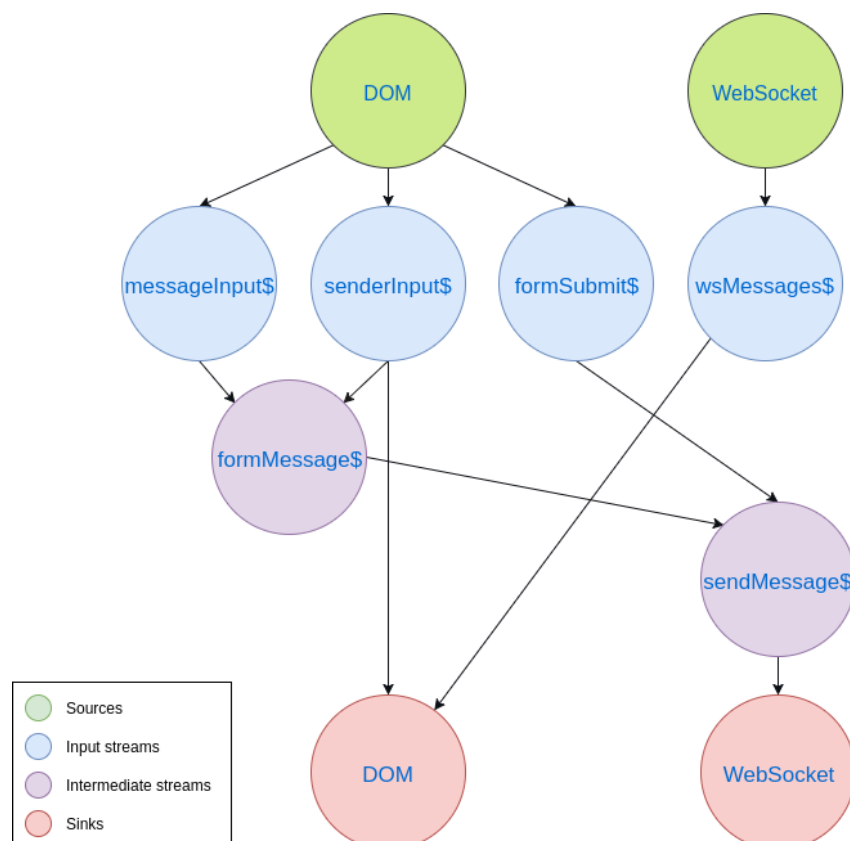


Figure 7.4: Dependency graph of the chat application

By using operators to mutate the input streams they are converted to the required format for the output streams. For the sake of readability intermediate streams are used. They are not required but are useful to make reasoning about the application easier. Finally the sinks are handled by the driver functions. The drivers handle all the I/O and side effects caused by the sinks. In the case of the chat application the DOM driver rerenders the DOM with the updated DOM and the WebSocket driver sends newly submitted messages over the WebSocket.

For elegant representation of the DOM in JavaScript, JSX is used. JSX is not a templating language but rather a syntax extension to JavaScript [49]. Because of this JSX comes with the full power of JavaScript. JSX was popularized by its use in React [49]. In the code we provide the DOM sink as a stream of JSX objects (see listing 7.2).

Listing 7.2: *Using JSX to define the DOM*

```
function main({ DOM, ... }) { // sources in function argument
  ...
  const vtree$ = Observable
    .combineLatest(wsMessages$.startWith([]), senderInput$.startWith(''))
    .map(([messages, me]) =>
      <div className="wrapper">
        <ul className="chat">
          { messages && messages.map(m =>
            <li className={`chat__entry${me === m.sender ? ' chat__entry--mine' : ''}`>
              <span className="chat__entry__message">{m.message}</span>
            </li>
          )}
        </ul>
      </div>
    );

  return { DOM: vtree$, ... }; // return sinks
}
```

In the code in listing 7.2 the path towards the DOM sink in the dependency graph shown in figure 7.4 can be recognized. The `wsMessages$` and the `senderInput$` streams are combined to render the current state of the application to the DOM. The `startWith()` operators are used to define the initial state.

The JSX in listing 7.2 is only an excerpt of the complete DOM. Only the logic to render the messages and determine if a message was sent by the current user is included. To render the messages from an array of objects the `Array.map()` function is used. This function maps the individual messages to JSX objects. To determine if the current user is the sender of a particular message a ternary is used. The complete source code of this implementation can be found in appendix A.

7.4 Frontend with imperative programming

7.4.1 Choice of technologies

When developers create web applications with imperative programming they frequently use libraries such as JQuery as an abstraction over vanilla JavaScript. JQuery makes things like HTML document traversal and manipulation, event handling, animation, and Ajax simpler [50]. While there are alternatives to JQuery like Dojo and Ext, JQuery has been and still is the most popular one by far.

Monolithic abstractions over JavaScript like JQuery have been losing popularity in recent years because of smaller and more modular utility libraries such as lodash. New features and APIs that are added to the JavaScript specification also diminish the need for JQuery. Last but not least more and more developers are migrating to more declarative frameworks and libraries to manipulate the DOM such as React.

For this application JQuery was chosen because it still sees the most usage compared to its competitors. React or Angular are not really options because they are not imperative programming frameworks.

Templating language

To render the application to the DOM it was decided to use a templating engine for convenience. Handlebars was chosen as the templating engine for the application. Handlebars provides some utility over Moustache which it is built upon but is still a very minimal templating engine that is much lighter than popular competitors such as Pug [51].

7.4.2 Architecture

The application uses a single `render()` function that is called whenever the application needs to rerender (see figure 7.5). The application listens to events from the DOM and from the WebSocket. These events then mutate objects that are stored in globally scoped mutable variables.

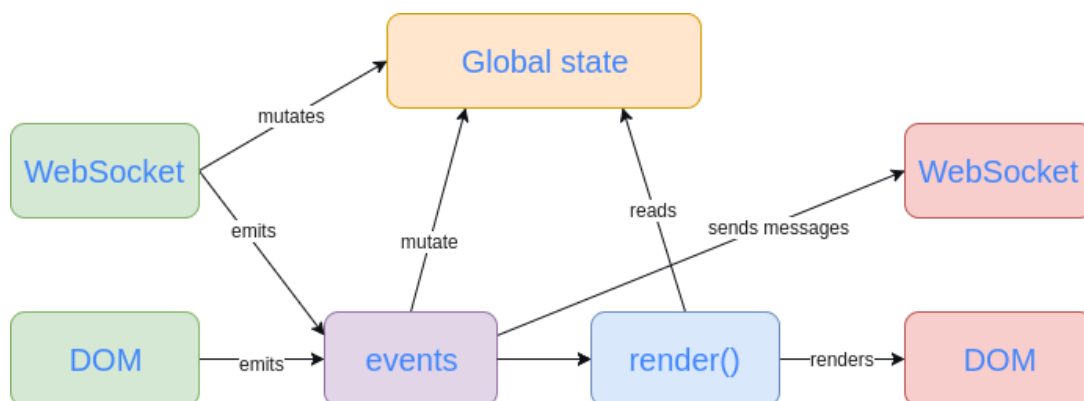


Figure 7.5: Diagram of the imperative application architecture

The event handlers from the `onmessage` event on the **WebSocket** and the `oninput` event from the

sender input call the `render()` function. Whenever the render function gets called it reads variables from the global state and rerenders the application in the DOM. The onsubmit event from the message form causes a new message to be sent over the WebSocket.

7.4.3 Implementation

This implementation uses event handlers to mutate a global state. This global state is instantiated as in listing 7.3.

Listing 7.3: *The global variables that make up the application state*

```
const template = Handlebars.compile(`
  <div class="wrapper">
    ...
  </div>
`);

const messages = [];
let lastMessage = '';
let me = '';
```

The `messages` variable gets mutated by an event handler on the `onmessage` event of the WebSocket. The `lastMessage` variable holds the last sent message so that it can be persisted as the input value after render. Lastly the `me` variable contains the current sender. This variable is mutated by the event handler on the `oninput` event of the text field.

This application architecture is completely different from the FRP implementation. In that implementation the event streams were composed into an unidirectional dataflow (see figure 7.4). In this implementation event handlers are set up separately. They then all mutate global objects to communicate their data to other parts of the application.

Render function

The `render()` function is implemented as shown in listing 7.4. Internally the render function compiles the Handlebars template with the new data. To have the necessary data the render function first loops over the messages to check which ones are from the current user. Finally it replaces the previous version of the application in the DOM with the newly compiled Handlebars template.

Listing 7.4: *Implementation of the render function*

```
const render = () => {
  for (let i = 0; i < messages.length; i++) {
    const message = messages[i];
    message.isMine = message.sender === me;
  }
  const data = { messages, me, lastMessage };
}
```

```
const htmlString = template(data);
$('#app').html(htmlString);
$('#sender').focus().val(me);
};
```

The last line of listing 7.4 covers a corner case where the sender input loses focus while the user is typing because of the rerender. Another corner case that needed to be covered was initial render. Because the `render()` function only gets called from events the application would not render until an event occurs. To solve this the `render()` function also has to be called on load. The complete source code of this implementation can be found in appendix B.

7.5 Comparison

In this chapter the two implementations from section 7.3 and section 7.4 are compared with various metrics. First a static analysis tool is used to analyze the application code. Second the performance of the two applications is compared. Finally some more subjective metrics like readability are discussed.

7.5.1 Static analysis

Static analysis of code is the analysis that is performed without executing the program [52]. This type of code analysis is performed on the source code. Static analysis is mostly used to ensure code quality across large code repositories. These tools are typically integrated into the build system and ran automatically for every commit [52]. In the case of this bachelor thesis it is used to compare the two implementations of the chat application.

Bundle size

The build system for both frontend implementations is identical. Both use a minimal Webpack setup (see appendix C). As a result the bundles are generated in the same way. Because of the small size of the application logic the bundle size depends mostly on the imported libraries. To compare and visualize these two bundles the source-map-explorer tool is used. This tool offers an interface to visualize JavaScript bundles and the relative sizes of their contents [53].

The JavaScript bundle size of the imperative implementation is approximately 165kB. As can be seen in figure 7.6, this bundle is split almost evenly between the two dependencies, JQuery and Handlebars.

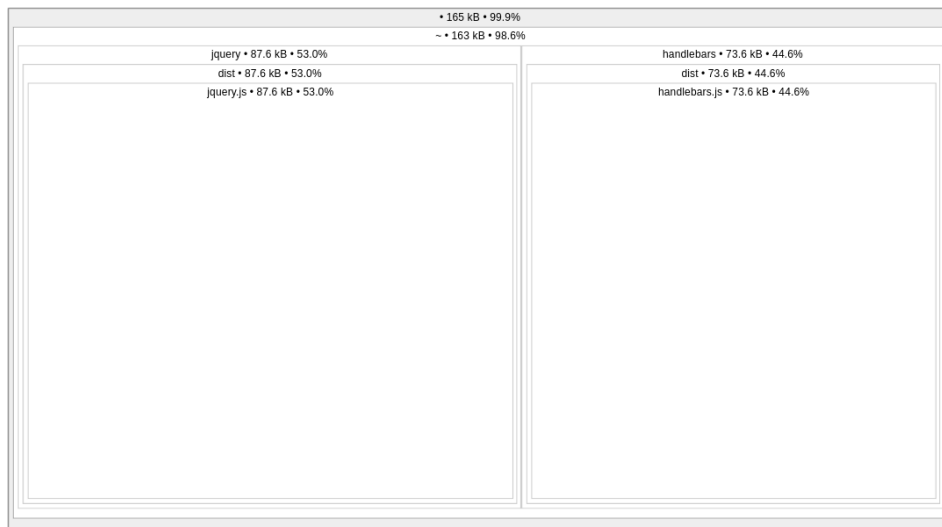


Figure 7.6: Visualisation of the JavaScript bundle of the imperative implementation

The bundle of the FRP implementation is somewhat more complicated. RxJS and Cycle.js are developed in a modular way, this is why importing parts of RxJS can be done with ES2015 module imports as seen in the first 10 lines of appendix A. In spite of that RxJS is still the largest dependency in this bundle by far. The total bundle size is 114kB (see figure 7.7).

Xstream is in this bundle because Cycle.js depends on it [42]. If xstream would be used in the application logic this dependency could be reused and more than 43kB could be saved by removing the dependency on RxJS. There are also some polyfills included in this bundle to be compatible with older browsers.

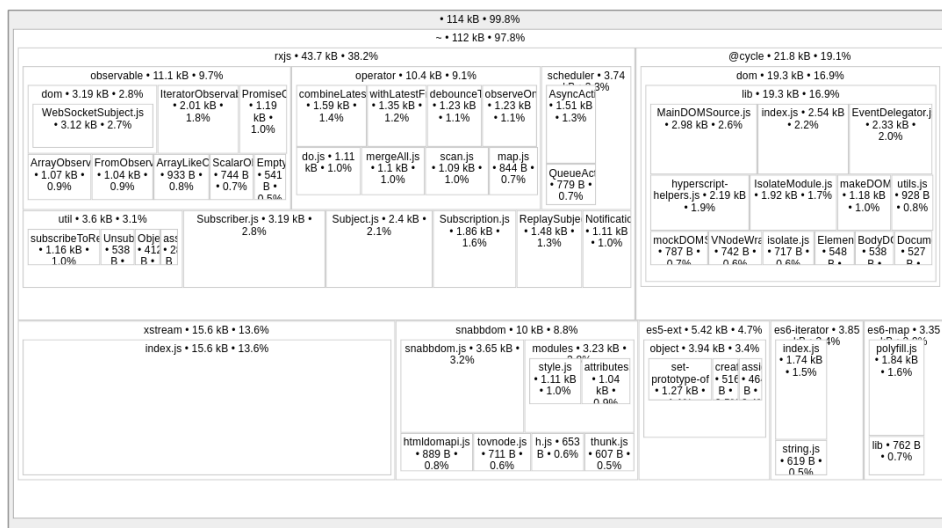


Figure 7.7: Visualisation of the JavaScript bundle of the FRP implementation

Tree shaking

The build script for this application uses Webpack v1.x which does not support tree shaking. Tree shaking

or dead-code elimination relies on the ES2015 module import/export syntax to eliminate unused code [54]. By migrating the build script to Webpack v2.x or Rollup, tree shaking can be leveraged to optimize the JavaScript bundle further [54][55].

Code length

Simple comparison

Code length can be evaluated by simply comparing the length of the program in terms of source lines of code (SLOC) and characters. In this calculation empty lines and import statements are excluded. Using this method the FRP implementation is the clear winner. It has only 45 SLOC compared to the imperative implementation's 76 SLOC. The FRP implementation also wins in terms of characters. It counts 1401 characters excluding spaces compared to the imperative implementation's 1797 characters.

Comparison by Halstead measurements

The comparison in the previous paragraph is valid but not very sophisticated. Character count for example can be heavily influenced by the length of identifiers. But longer identifiers for variables or functions are not necessarily bad, they might make a program more legible. To compare more sophisticated metrics like number of operators and operands Halstead complexity measurements can be used [56].

To generate these measurements for this application Plato is used. Plato is a tool to report and visualize measurements from static analysis on JavaScript code. These measurements include cyclomatic code complexity and Halstead complexity measures [57]. The full reports from Plato can be found in appendix D and E in JSON format. The relevant Halstead measurements for code length are displayed in table 7.1.

| Metric | Imperative implementation | FRP implementation |
|---------------------|----------------------------------|---------------------------|
| Number of operands | 129 | 136 |
| Distinct operands | 57 | 72 |
| Number of operators | 122 | 116 |
| Distinct operators | 20 | 17 |
| Program length | 251 | 252 |
| Physical SLOC | 87 | 58 |
| Logical SLOC | 39 | 15 |
| Volume | 1573 | 1631 |
| Vocabulary | 77 | 89 |

Table 7.1: *Halstead measurements for both implementations (Lower is better)*

The number of operands and operators is almost equal for both implementations. The FRP implementation has slightly more operands and slightly less operators compared to the imperative implementation. As a result the Halstead program length (sum of total operands and operators) is as good as equal for both implementations.

Logical SLOC is an interesting measurement. It counts the number of imperative statements and not the number of lines like physical SLOC [58]. Unsurprisingly the FRP implementation has less imperative statements than the imperative implementation. In fact the Logical SLOC of the FRP implementation is less than half of the logical SLOC of the imperative implementation.

The FRP implementation has more vocabulary and a higher volume than the imperative implementation. The Halstead vocabulary is simply the sum of the number of distinct operators and operands [59]. This is higher for the FRP implementation because of the high number of distinct operators it introduces. This might contribute to the steep learning curve for learning FRP. The Halstead volume is calculated from the program length and the vocabulary [59]. This measurement being higher for the FRP implementation is simply a result of the volume being higher.

Cyclomatic complexity

Cyclomatic complexity is a source code complexity metric to measure the number of independent paths execution can take through a program [60]. Cyclomatic complexity is calculated by constructing the control flow graph of a program. For a single function the cyclomatic complexity is then calculated by the following formula: $M = E - N + 2$ [60]. In that formula M stands for the cyclomatic complexity, E for the

number of edges of the control flow graph and N for the number of nodes in the graph. Figure 7.8 shows the control flow graph of a simple example program shown in listing 7.5.

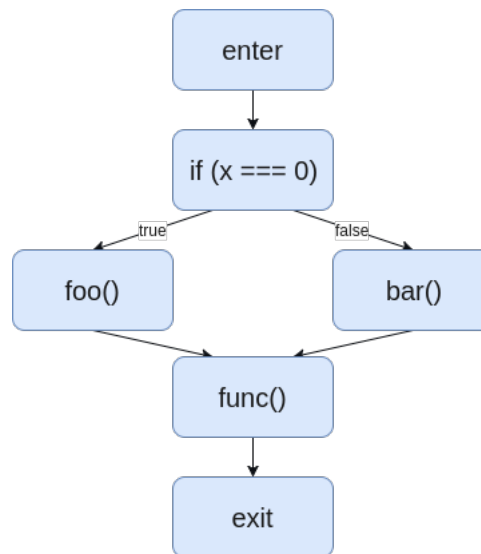


Figure 7.8: Control flow graph of a simple program (see listing 7.5)

Using the aforementioned formula on this function results in a cyclomatic complexity of 2 ($6 - 6 + 2$). This makes sense because there are only two possible paths through this code. One where the if statement is true and one where it is false.

Listing 7.5: Example code to illustrate the calculation of cyclomatic complexity

```
function run(x) {  
  if (x === 0) {  
    foo();  
  } else {  
    bar();  
  }  
  func();  
}
```

The cyclomatic complexities of the two implementations are both relatively low. The FRP implementation has a cyclomatic complexity of 3 and a density of 20. The density of cyclomatic complexity is calculated by comparing it with the SLOC [60]. The imperative implementation has a higher complexity (5) but a lower density (12.8).

Plato metrics

The Plato tool provides a number of different metrics to indicate code quality. Most of them are Halstead complexity measurements. Table 7.2 shows a selection of the most relevant measurements for both implementations.

| Metric | Imperative implementation | FRP implementation |
|-----------------------|---------------------------|--------------------|
| Difficulty (Halstead) | 23 | 16 |
| Effort (Halstead) | 35599 | 26201 |
| Time (Halstead) | 1977 | 1455 |
| Maintainability | 70 | 83 |

Table 7.2: *Plato measurements for both implementations*

The Halstead difficulty measures how difficult it is to write a program. This measurement is calculated from the number of operators and operands [59]. According to this measurement the FRP implementation is significantly less difficult to program.

The Halstead effort is the product of the Halstead difficulty and volume. The Halstead time is a weighted version of the effort [56]. It is meant as an approximation of the time it would take to write a program in seconds. Since the difficulty and the volume of the imperative implementation are both higher than those of the FRP implementation the effort and time measurements are also higher. In fact this measurement indicates that the imperative implementation requires 26% more effort to program.

Maintainability is a logarithmic scale from negative infinity to 171 introduced by Paul Oman and Jack Hagemester in 1991 [56]. It is calculated from the logical SLOC, the cyclomatic complexity and the Halstead effort [56]. All of these measurements are worse for the imperative implementation. As a result it scores significantly worse than the FRP implementation for maintainability.

7.5.2 Performance

To compare the performance of these two implementations two usage scenarios are evaluated. To measure performance Chrome Devtools is used.

Initial load

The imperative implementation loads significantly faster than the FRP implementation (see figure 7.9 and 7.10). The imperative application loads in 182ms whereas the FRP application loads in 209ms. These times are measured from the time of the initial HTTP request until the DOMContentLoaded event. This result is surprising considering the bundle of the imperative application is bigger than the FRP bundle (see section 7.5.1).

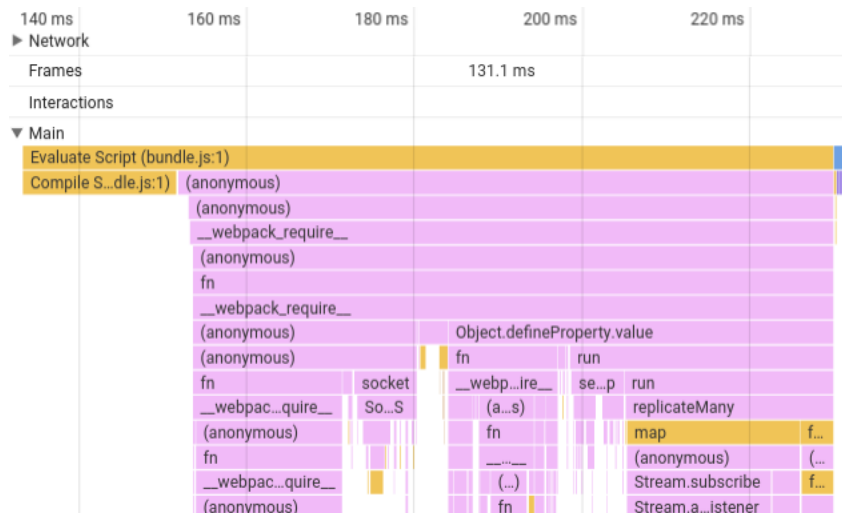


Figure 7.9: Devtools timeline of initial load in the FRP application

This difference can be explained by the internal work Cycle.js is doing on startup. Cycle.js needs to initialize the source and sink streams for the application as well as its DOM driver. The Cycle.js DOM driver uses a virtual DOM implementation to optimize the changes it does to the DOM. This means that it has to load a representation of the DOM into memory on initial load. After all that work is completed it can run the `main()` function to do the initial render of the application.

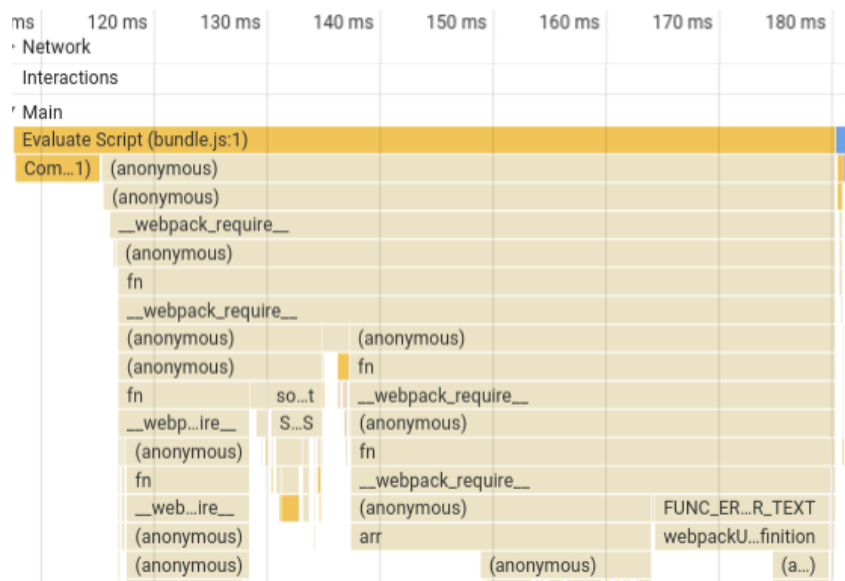


Figure 7.10: Devtools timeline of initial load in the imperative application

The work done in the imperative application is considerably less complex. The only thing the application does on initial load is instantiating some variables to hold the global state and then calling the `render()` function. The render function then runs the templating engine with the data from the initial global state and injects it into the DOM. This application keeps no virtual representation of the DOM in memory.

Memory consumption is almost equal for both implementations (see figure 7.11 and 7.12). Interestingly

the imperative application mounts 114 more DOM nodes than the FRP implementation even though HTML markup is identical for both applications.

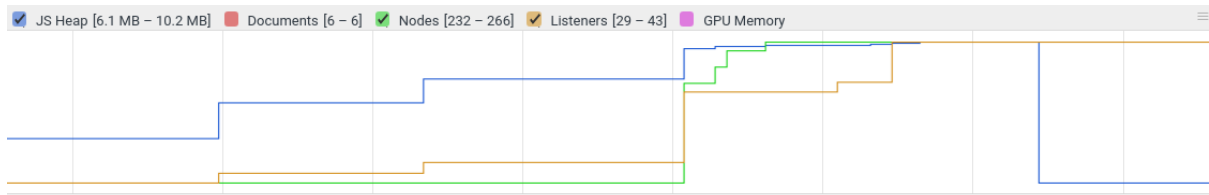


Figure 7.11: Memory allocation when loading the FRP application

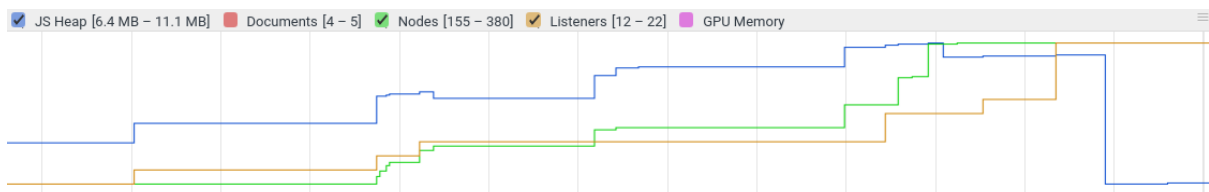


Figure 7.12: Memory allocation when loading the imperative application

The difference can be explained by the optimization of the Cycle.js DOM driver. This driver calculates the difference of the current DOM state and the new state using its virtual DOM. As a result it only replaces the nodes in the DOM that have changed. The imperative implementation does not have these optimizations and just replaces all the DOM nodes in the application. The reason why the application is rerendered a few times on load is because the server sends welcome messages over the WebSocket.

Adding a message

In this scenario the applications need to debounce the submit event of the message form and then send the message over the WebSocket. When the server broadcasts this message to them they need to render it in the DOM. In figure 7.13 and 7.14 the Chrome Devtools timeline for both applications is shown.

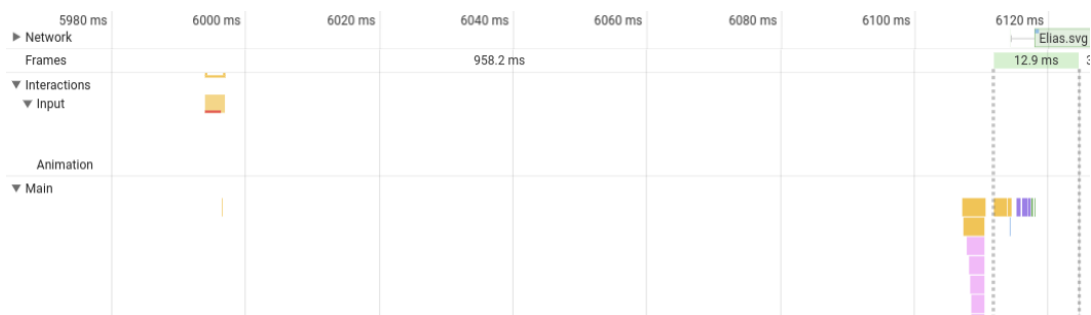


Figure 7.13: Devtools timeline when adding a message in the FRP application

The first yellow section that is visible in the input row on both timelines is the submit event from the form. After the event a delay follows because of the debounce function. The sending of the message

over the WebSocket is very fast in both implementations but the imperative implementation is faster. The imperative implementation does it in 1.8ms and the FRP version in 3.5ms. That is a result of the imperative implementation using the native WebSocket API directly. The FRP implementation uses an abstraction layer from RxJS on top of that API [47].

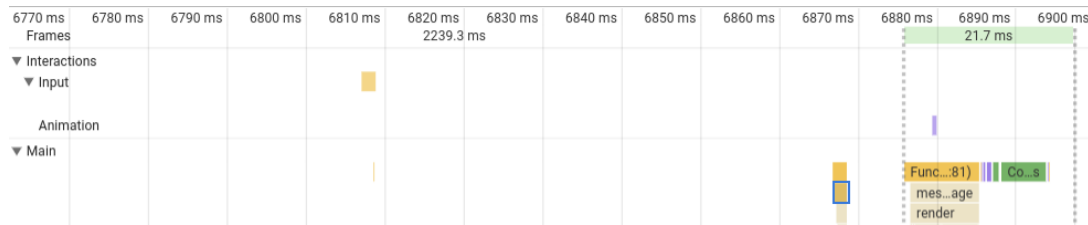


Figure 7.14: Devtools timeline when adding a message in the imperative application

Receiving and rendering the message is considerably slower in the imperative application. It takes 21.7ms to receive the message and render it to the DOM. 21.7ms of processing time means that the browser momentarily dipped below 60fps. This could cause stuttering animation in the application. The FRP application only takes 12.9ms and stays comfortably above 60fps.

This difference can be explained by the more efficient DOM rendering done internally by Cycle.js. Because of its virtual DOM it only replaces the DOM nodes that need to be replaced. A lot of these performance measurements depend heavily on framework specific implementations. In section 7.5.4 this is discussed further and a third implementation in vanilla JavaScript without any dependencies is compared.

7.5.3 Readability

Readability of a program is arguably more important than performance or efficiency. An argument for this is that programmers spend a lot of time reading code, arguably more time than they spend writing it [61]. Developers read code when debugging, doing peer reviews and learning new libraries. Because applications are almost always written by multiple developers it is important that code is readable and understandable. The consensus in the industry is that optimizing for readability is more beneficial than optimizing for performance [61].

In general there are two factors that make code less readable. The code can be hard to understand or it can be hard to follow. Code can be hard to understand if it tries to be too clever or does undocumented low-level optimizations. Code is hard to follow if there is no clear structure or architecture. It can also be hard to follow if the code execution is disconnected. In this case disconnected means that it is not possible to step through the code and follow the order of execution [61].

The FRP implementation follows an unidirectional dataflow. This architecture allows the developer to see the program as a pipeline. Data flows through each pipe and is mutated and combined with data from different pipes. This makes an application easy to reason about and easy to visualize with a marble diagram or a dependency graph. The catch is that being able to understand this code requires knowledge of the FRP paradigm and of the API of an FRP library.

The imperative implementation mutates a global state through multiple event handlers. This makes this implementation harder to follow. Different event handlers might be mutating the same object and it is hard to deal with this concurrency. When trying to follow this code the reader needs to constantly jump between global variables and the functions they are mutated from. Composing these events into a single flow is not possible without the FRP paradigm. Especially when the application deals with real-time data such as data from WebSockets it becomes cumbersome to manage all these events.

7.5.4 Influence of dependencies on results

Choice of frameworks and libraries has a big influence on the comparison done in this chapter. Especially the performance section depends almost entirely on the implementation of the library. Because of this a third implementation is developed. This implementation also uses imperative programming but uses no dependencies. It would be useful to also have an FRP implementation without any dependency to compare with but unfortunately there is no support for an observable data structure built in to JavaScript (yet [34]). The complete implementation in vanilla JavaScript is available in appendix F. The most important metrics from static and performance analysis are shown in table 7.3.

| Metric | Imperative implementation (vanilla JS) | Imperative implementation (jQuery) | FRP implementation |
|-----------------------------|---|---|-------------------------------|
| Bundle size | 1.8kB | 165kB | 114kB |
| Maintainability | 69 | 70 | 83 |
| Program length (Halstead) | 469 | 251 | 252 |
| Time (Halstead) | 5860 | 1977 | 1455 |
| Logical SLOC | 78 | 39 | 15 |
| Cyclomatic complexity | 9 | 5 | 3 |
| Initial load duration | 147ms | 182ms | 209ms |
| Send message duration | 0.4ms | 1.8ms | 3.5ms |
| Render new message duration | 9.5ms | 21.7ms | 12.9ms |

Table 7.3: Summary of metrics from static and performance analysis

The vanilla JavaScript implementation bundle is only a fraction of the size of the bundle of the other two implementations because it has no dependencies. On all performance metrics it is faster than

the other two. This makes sense because it is directly using the native browsers APIs instead of abstractions.

All these size and performance winnings come at the cost of extra code complexity. The cyclomatic complexity is almost twice as high as the imperative implementation with JQuery. All Halstead metrics are also considerably worse for this implementation. The Halstead time is almost 3 times as high as the JQuery implementation and more than 4 times as high as the FRP implementation. This indicates that this version of the application would take 4 times longer to write than the FRP implementation. This increased overall complexity and length also has a negative impact on readability.

Conclusion

As the web as a platform is transitioning more towards applications instead of documents JavaScript usage is climbing. This trend makes the core challenges of working with JavaScript to make applications more prominent. One of the biggest challenges is managing asynchronous code in JavaScript. Callbacks are the most basic technique to manage asynchronous code but they are hard to compose and quickly result in unreadable code and bad practices. Promises and `async/await` are big improvements but they still lack an advanced API to compose and mutate asynchronous data. Functional reactive programming is a programming paradigm that solves this challenge by implementing the concept of asynchronous data streams.

As the results of the case study indicate functional reactive programming has a significant positive impact on complexity, readability and maintainability of an application. Especially in real-time applications data streams are an excellent concept to handle real-time data in an efficient manner. The JavaScript API uses event handlers with callback functions. The problem with those is that they can not be combined and composed like streams can. They have to store state in mutable variables to communicate. This works for a small application but quickly becomes unmanageable and hard to read.

The functional reactive programming paradigm also introduces a new way to think about application architecture. Applications can be seen as a pipeline of data. Developers declare how the data flows through the pipes and how it is changed along the way. This declarative approach to application development decreases the number of imperative statements. As a result readability and maintainability of the application is increased.

Functional reactive programming as a paradigm brings some challenges with it however. Even though library documentation and introductions to FRP have become better, learning to think reactive is still time consuming. This can be a challenge since application development today is mostly done in teams. Getting an entire team to be confident in programming with FRP can be hard and might not be worth the effort for smaller applications.

Since JavaScript does not provide a data structure for streams in the core language, developers still need to rely on libraries at the time of writing. As any library or layer of abstraction would, this heavily increases the size of the JavaScript bundle served to users. This can be an important argument not to use functional reactive programming as more people in developing countries gain access to the web. This extra layer of abstraction also causes decreased performance on runtime. However the results from the case study show us that this hit in performance is not very large.

Functional reactive programming is a great abstraction for any application that deals with a large number of asynchronous events. Since applications that use data from real-time sources such as WebSockets are inherently heavy on events, modeling them as streams using FRP can be a big advantage. It remains important to know the application's users and define the priorities. If a web application is targeted towards a demographic with poor connectivity or if performance is a priority it might not be a good decision to adopt functional reactive programming. For most applications though optimizing for readability and maintainability first is advised. It might even result in better performance as the size and complexity of the application increase.

References

- [1] N. Lomas. (2016). Showpad closes \$50m series c to keep growing its saas sales productivity platform, [Online]. Available: <https://techcrunch.com/2016/05/19/showpad-closes-50m-series-c-to-keep-growing-its-saas-sales-productivity-platform/> (visited on 2017-03-02).
- [2] Showpad. (2017). Smarter sales enablement, [Online]. Available: <https://www.showpad.com/who-we-are/> (visited on 2017-03-02).
- [3] J. M. Lemkin. (2015). Coming to the u.s.: Learnings at ~\$10m arr from showpad and conversocial, [Online]. Available: <https://saastr.quora.com/Coming-to-the-U-S-Learnings-at-10m-ARR-From-Showpad-and-Conversocial> (visited on 2017-03-06).
- [4] E. Meire. (2017). Rxjs recipes, [Online]. Available: <http://slides.com/eliasmeire/rx#/> (visited on 2017-05-15).
- [5] D. Radigan. (2016). How the kanban methodology applies to software development, [Online]. Available: <https://www.atlassian.com/agile/kanban> (visited on 2017-03-01).
- [6] N.N. (2017). Gears17, [Online]. Available: <http://gears.gent> (visited on 2017-03-15).
- [7] N.N. (2016). Showtime16, [Online]. Available: <http://showtime.showpad.com> (visited on 2017-03-13).
- [10] N.N. (2017). Websockets, [Online]. Available: <https://en.wikipedia.org/wiki/WebSocket> (visited on 2017-03-18).
- [11] N.N. (2017). Websockets, [Online]. Available: <http://caniuse.com/#feat=websockets> (visited on 2017-03-18).
- [14] N.N. (2017). Webrtc peer-to-peer connections, [Online]. Available: <http://caniuse.com/#feat=rtcpeerconnection> (visited on 2017-03-20).
- [15] N.N. (2017). Webrtc: The future of web games, [Online]. Available: <https://news.ycombinator.com/item?id=13266692> (visited on 2017-03-18).
- [22] N.N. (2017). Observer, [Online]. Available: https://sourcemaking.com/design_patterns/observer (visited on 2017-03-19).
- [23] N.N. (2017). Observable instance method delay, [Online]. Available: <http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-delay> (visited on 2017-04-22).
- [25] P. Irish. (2017). Devtools: State of the union 2017 (google i/o '17), [Online]. Available: <https://www.youtube.com/watch?v=PjjlwAvV8Jg> (visited on 2017-05-20).
- [28] A. Susiripala. (2015). Top es2015 features in 15 minutes, [Online]. Available: <https://kadi.io/blog/other/top-es2015-features-in-15-minutes> (visited on 2017-03-10).
- [29] B. McCormick. (2015). Es6 patterns: Converting callbacks to promises, [Online]. Available: <https://benmccormick.org/2015/12/30/es6-patterns-converting-callbacks-to-promises/> (visited on 2017-04-02).

- [30] N.N. (2016). Promise.all(), [Online]. Available: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise/all (visited on 2017-05-02).
- [31] N.N. (2017). Async functions, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function (visited on 2017-04-13).
- [32] A. Agrawal. (2017). What promises do that observables can't, [Online]. Available: <http://moduscreate.com/observables-and-promises> (visited on 2017-04-15).
- [34] N.N. (2017). Observable proposal, [Online]. Available: <https://github.com/tc39/proposal-observable> (visited on 2017-03-19).
- [36] E. Meire, *Ws-chat*, 2017. [Online]. Available: <https://github.com/eliasmeire/ws-chat> (visited on 2017-05-20).
- [39] N.N. (2017). Gears17, [Online]. Available: <http://gears.gent> (visited on 2017-03-15).
- [40] N.N. (2017). Gears17, [Online]. Available: <https://www.npmjs.com/package/most> (visited on 2017-05-02).
- [41] A. Staltz. (2017). Xstream, [Online]. Available: <https://www.npmjs.com/package/xstream> (visited on 2017-05-02).
- [43] A. Staltz. (2017). André staltz, [Online]. Available: <https://staltz.com/about.html> (visited on 2017-05-01).
- [45] E. Meire. (2017). Cycle-ws-driver, [Online]. Available: <https://www.npmjs.com/package/cycle-ws-driver> (visited on 2017-05-20).
- [46] E. Meire. (2017). Cycle-ws-driver, [Online]. Available: <https://github.com/eliasmeire/cycle-ws-driver> (visited on 2017-05-21).
- [47] N.N. (2017). Rxjs/websocketsubject.ts at master · reactiveX/rxjs, [Online]. Available: <https://github.com/ReactiveX/rxjs/blob/master/src/observable/dom/WebSocketSubject.ts> (visited on 2017-05-05).
- [48] N.N. (2017). Cyclejs/storage - a cycle.js driver for using localStorage and sessionStorage, [Online]. Available: <https://github.com/cyclejs/storage> (visited on 2017-05-10).
- [50] N.N. (2017). JQuery, [Online]. Available: <https://jquery.com/> (visited on 2017-05-01).
- [51] N.N. (2017). Handlebars.js: Minimal templating on steroids, [Online]. Available: <http://handlebarsjs.com/> (visited on 2017-05-05).
- [52] N.N. (2017). Static program analysis, [Online]. Available: https://en.wikipedia.org/wiki/Static_program_analysis (visited on 2017-04-14).
- [53] D. Vanderkam. (2017). Source-map-explorer, [Online]. Available: <https://www.npmjs.com/package/source-map-explorer> (visited on 2017-04-28).
- [54] N.N. (2017). Tree shaking, [Online]. Available: <https://webpack.js.org/guides/tree-shaking/> (visited on 2017-05-10).
- [55] N.N. (2017). Rollup.js - guide, [Online]. Available: <https://rollupjs.org/> (visited on 2017-05-10).
- [56] H. Zuse. (2005). Resolving the msyteries of the halstead measures, [Online]. Available: <http://www.horst-zuse.homepage.t-online.de/z-halstead-final-05-1.pdf> (visited on 2017-05-11).
- [57] N.N. (2016). Es-analysis/plato - javascript source code visualization, static analysis, and complexity tool, [Online]. Available: <https://github.com/es-analysis/plato> (visited on 2017-05-12).

- [58] R. Gómez. (2013). How complex are todomvc implementations | coderstats blog, [Online]. Available: <http://blog.coderstats.net/todomvc-complexity/> (visited on 2017-05-12).
- [59] N.N. (2014). Ibm knowledge center | halstead metrics, [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm (visited on 2017-05-11).
- [60] P. Anderson. (2016). Your cyclomatic complexity may be wrong, [Online]. Available: <http://blogs.grammatech.com/your-cyclomatic-complexity-may-be-wrong> (visited on 2017-05-11).

Bibliography

- [8] P. Lubbers. (2014). Html5 websocket: A quantum leap in scalability for the web, [Online]. Available: <https://websocket.org/quantum.html> (visited on 2017-03-13).
- [9] J. Hanson. (2014). Http long polling, [Online]. Available: <https://www.pubnub.com/blog/2014-12-01-http-long-polling/> (visited on 2017-03-06).
- [12] S. Dutton. (2014). Getting started with webRTC, [Online]. Available: <https://www.html5rocks.com/en/tutorials/webRTC/basics> (visited on 2017-03-18).
- [13] D. Ristic. (2014). WebRTC data channels, [Online]. Available: <https://www.html5rocks.com/en/tutorials/webRTC/datachannels/> (visited on 2017-03-22).
- [16] N.N. (2017). Functional reactive programming, [Online]. Available: https://en.wikipedia.org/wiki/Functional_reactive_programming (visited on 2017-03-03).
- [17] N.N. (2017). Functional reactive programming, [Online]. Available: https://wiki.haskell.org/Functional_Reactive_Programming (visited on 2017-03-02).
- [18] M. Fogus, *Functional javascript*. O'Reilly Media, 2013, ISBN: 9781449360726.
- [19] N.N. (2016). Lambda calculus, [Online]. Available: https://en.wikipedia.org/wiki/Lambda_calculus (visited on 2017-02-24).
- [20] M. R. Cook. (2016). A practical introduction to functional programming, [Online]. Available: <https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming> (visited on 2017-02-24).
- [21] A. Staltz. (2016). The introduction to reactive programming you've been missing, [Online]. Available: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754> (visited on 2017-02-22).
- [24] A. Staltz. (2015). How to debug rxjs code, [Online]. Available: <https://staltz.com/how-to-debug-rxjs-code.html> (visited on 2017-04-30).
- [26] B. Lesh. (2016). Hot vs cold observables, [Online]. Available: <https://medium.com/@benlesh/hot-vs-cold-observables-f8094ed53339> (visited on 2017-05-10).
- [27] B. Lesh. (2016). On the subject of subjects (in rxjs), [Online]. Available: <https://medium.com/@benlesh/on-the-subject-of-subjects-in-rxjs-2b08b7198b93> (visited on 2017-05-11).
- [33] B. Lesh. (2016). Netflix javascript talks - rxjs version 5, [Online]. Available: <https://www.youtube.com/watch?v=C0viCoUtwx4> (visited on 2017-04-03).
- [35] E. Czaplicki and S. Chong. (2013). Asynchronous functional reactive programming for guis, [Online]. Available: <http://people.seas.harvard.edu/~chong/pubs/pldi13-elm.pdf> (visited on 2017-04-03).
- [37] N.N. (2017). Uwebsockets/uwebsockets, [Online]. Available: <https://github.com/uWebSockets/uWebSockets> (visited on 2017-04-13).
- [38] N.N. (2017). Reactivex, [Online]. Available: <http://reactivex.io/> (visited on 2017-04-17).
- [42] N.N. (2017). Cycle.js, [Online]. Available: <https://cycle.js.org/> (visited on 2017-04-21).

- [44] N.N. (2017). Cycle.js - drivers, [Online]. Available: <https://cycle.js.org/drivers.html> (visited on 2017-04-21).
- [49] N.N. (2017). Introducing jsx, [Online]. Available: <https://facebook.github.io/react/docs/introducing-jsx.html> (visited on 2017-04-14).
- [61] V. Simonov. (2014). Optimize for readability first, [Online]. Available: <http://va.lent.in/optimize-for-readability-first/> (visited on 2017-05-13).

Appendices

A FRP frontend application logic

```
1  import { Observable } from 'rxjs/Observable';
2  import 'rxjs/add/observable/dom/webSocket';
3  import 'rxjs/add/observable/combineLatest';
4  import 'rxjs/add/operator/do';
5  import 'rxjs/add/operator/map';
6  import 'rxjs/add/operator/scan';
7  import 'rxjs/add/operator/withLatestFrom';
8  import 'rxjs/add/operator/debounceTime';
9  import 'rxjs/add/operator/startWith';
10 import { html } from 'snabbdom-jsx';
11
12 export function App ({ DOM, ws }) {
13   const wsMessages$ = ws.scan((acc, m) => [...acc, m], []);
14   const formSubmit$ = DOM.select('#form').events('submit').do(e =>
15     e.preventDefault());
16   const senderInput$ = DOM.select('#sender').events('input').map(e =>
17     e.target.value);
18   const messageInput$ = DOM.select('#message').events('input').map(e =>
19     e.target.value);
20   const formMessage$ = Observable.combineLatest(senderInput$, messageInput$)
21     .map(([sender, message]) => ({ sender, message }));
22
23   const sendMessage$ = formSubmit$
24     .debounceTime(200)
25     .withLatestFrom(formMessage$, (event, message) => JSON.stringify(message));
26   const vtree$ = Observable
27     .combineLatest(wsMessages$.startWith([]), senderInput$.startWith(''))
28     .map(([messages, me]) =>
29       <div className="wrapper">
30         <ul className="chat">
31           { messages &&
32             messages.map(m =>
33               <li className={`chat__entry ${me === m.sender &&
34                 'chat__entry--mine`} `}>
35                 <img
36                   className="chat__entry__avatar"
37                   src={`https://api.adorable.io/avatars/32/${m.sender}.svg`}
38                   title={m.sender}
39                   alt={m.sender}
40                 />
41                 <span className="chat__entry__message">{m.message}</span>
```

```

38         </li>
39     })
40 </ul>
41 <div className="form__container">
42     <form id="form" action="">
43         <div className="form__inputs">
44             <input className="form__input" id="sender" type="text"
45                 placeholder="Your name"/>
46             <input className="form__input" id="message" type="text"
47                 placeholder="Type a message..." />
48         </div>
49         <div className="form__submit">
50             <input className="button" type="submit" id="button__send"
51                 value="Send" />
52         </div>
53     </form>
54 </div>
55 );
56
57 return { DOM: vtree$, ws: sendMessage$ };
58 }

```

B Imperative frontend application logic

```
1  import $ from 'jquery';
2  import Handlebars from 'handlebars';
3  import debounce from 'lodash.debounce';
4
5  Handlebars.registerHelper('equal', (val1, val2, options) => {
6    val1 === val2 ? options.fn(this) : options.inverse(this);
7  });
8
9  $(document).ready(() => {
10    const template = Handlebars.compile(`
11      <div class="wrapper">
12        <ul class="chat">
13          {{#each messages}}
14            {{#if this.isMine}}
15              <li class="chat__entry chat__entry--mine">
16                <span class="chat__entry__message">{{this.message}}</span>
17              </li>
18            {{else}}
19              <li class="chat__entry">
20                
26                <span class="chat__entry__message">{{this.message}}</span>
27              </li>
28            {{/if}}
29          {{/each}}
30        </ul>
31        <div class="form__container">
32          <form id="form">
33            <div class="form__inputs">
34              <input class="form__input" id="sender" type="text"
35                placeholder="Your name" />
36              <input class="form__input" id="message" type="text"
37                value="{{lastMessage}}" placeholder="Type a message..." />
38            </div>
39            <div class="form__submit">
40              <input class="button" type="submit" id="button__send" value="Send"
41                />
42            </div>
43          </form>
44        </div>
45      </div>
46    `);
47    const wrapper = document.querySelector('.wrapper');
48    wrapper.innerHTML = template(messages);
49    const form = document.querySelector('#form');
50    const sender = document.querySelector('#sender');
51    const message = document.querySelector('#message');
52    const button = document.querySelector('#button__send');
53    button.addEventListener('click', () => {
54      const name = sender.value;
55      const msg = message.value;
56      if (name.trim() < 1 || msg.trim() < 1) return;
57      messages.push({ sender: name, message: msg, isMine: true });
58      messages.sort((a, b) => a.timestamp - b.timestamp);
59      wrapper.innerHTML = template(messages);
60      message.value = '';
61      lastMessage = msg;
62    });
63    debounce(() => {
64      messages.push({ sender: name, message: msg, isMine: false });
65      messages.sort((a, b) => a.timestamp - b.timestamp);
66      wrapper.innerHTML = template(messages);
67    }, 500)();
68  });
```

```

39         </div>
40     </form>
41 </div>
42 </div>
43 ');
44 const messageWebsocket = new WebSocket('ws://localhost:8001');
45 const messages = [];
46 let lastMessage = '';
47 let me = '';
48
49 const render = () => {
50     for (let i = 0; i < messages.length; i++) {
51         const message = messages[i];
52         message.isMine = message.sender === me;
53     }
54     const data = { messages, me, lastMessage };
55     const htmlString = template(data);
56     $('#app').html(htmlString);
57     $('#sender').focus().val(me);
58 };
59
60 render();
61
62 $('#app').on('submit', '#form', (event) => {
63     event.preventDefault();
64     debounce(() => {
65         const message = $('#message').val();
66         const sender = $('#sender').val();
67
68         if (message.trim().length > 0 && sender.trim().length > 0) {
69             const newMessage = { message, sender };
70             lastMessage = message;
71             messageWebsocket.send(JSON.stringify(newMessage));
72         }
73     }, 200)();
74 });
75
76 $('#app').on('input', '#sender', (event) => {
77     me = event.target.value;
78     render();
79 });
80
81 messageWebsocket.onmessage = (event) => {
82     const message = JSON.parse(event.data);
83     messages.push(message);
84     render();
85 };

```


C Build script

```
1  'use strict'
2
3  const fs = require('fs-extra')
4  const path = require('path')
5  const mkdirp = require('mkdirp')
6  const webpack = require('webpack')
7  const ProgressBarPlugin = require('progress-bar-webpack-plugin')
8
9  const buildPath = path.join(process.cwd(), 'build')
10 const publicPath = path.join(process.cwd(), 'public')
11
12 mkdirp.sync(buildPath)
13
14 const compiler = webpack({
15   entry: [
16     './src/'
17   ],
18   output: {
19     filename: 'bundle.js',
20     path: './public/'
21   },
22   devtool: 'source-map',
23   module: {
24     loaders: [
25       {
26         test: /\.js$/,
27         loader: 'babel',
28         query: {
29           presets: ['es2015']
30         },
31         exclude: /node_modules/
32       }
33     ],
34   },
35   plugins: [
36     new ProgressBarPlugin(),
37     new webpack.optimize.UglifyJsPlugin({minimize: true})
38   ]
39 })
40
41 compiler.run((err, stats) => {
```

```
42     if (err) {  
43         console.log(err)  
44     } else {  
45         fs.copySync(publicPath, buildPath)  
46     }  
47 })
```

D Plato report for FRP implementation

```
1  {
2    "summary": {
3      "total": {
4        "eslint": 0,
5        "sloc": 58,
6        "maintainability": 82.635
7      },
8      "average": {
9        "sloc": 58,
10       "maintainability": "82.64",
11       "eslint": "0.00"
12     }
13   },
14   "reports": [
15     {
16       "info": {
17         "file": "frontend/frp-cyclejs/src/app.js",
18         "fileShort": "frontend/frp-cyclejs/src/app.js",
19         "fileSafe": "frontend_frp_cyclejs_src_app_js",
20         "link": "files/frontend_frp_cyclejs_src_app_js/index.html"
21       },
22       "eslint": {
23         "messages": 0
24       },
25       "complexity": {
26         "aggregate": {
27           "complexity": {
28             "cyclomatic": 3,
29             "cyclomaticDensity": 20,
30             "halstead": {
31               "bugs": 0.544,
32               "difficulty": 16.056,
33               "effort": 26200.817,
34               "length": 252,
35               "time": 1455.601,
36               "vocabulary": 89,
37               "volume": 1631.885,
38               "operands": {
39                 "distinct": 72,
40                 "total": 136,
41                 "identifiers": [
```

```
42         "__stripped__"
43     ]
44 },
45     "operators": {
46         "distinct": 17,
47         "total": 116,
48         "identifiers": [
49             "__stripped__"
50         ]
51     }
52 },
53     "params": 11,
54     "sloc": {
55         "logical": 15,
56         "physical": 58
57     }
58 }
59 },
60     "module": "frontend/frp-cyclejs/src/app.js",
61     "maintainability": 82.635
62 }
63 }
64 ]
65 }
```

E Plato report for imperative programming implementation

```
1 {
2   "summary": {
3     "total": {
4       "eslint": 0,
5       "sloc": 87,
6       "maintainability": 70.429
7     },
8     "average": {
9       "sloc": 87,
10      "maintainability": "70.43",
11      "eslint": "0.00"
12    }
13  },
14  "reports": [
15    {
16      "info": {
17        "file": "frontend/imperative/src/app.js",
18        "fileShort": "frontend/imperative/src/app.js",
19        "fileSafe": "frontend_imperative_src_app_js",
20        "link": "files/frontend_imperative_src_app_js/index.html"
21      },
22      "eslint": {
23        "messages": 0
24      },
25      "complexity": {
26        "aggregate": {
27          "complexity": {
28            "cyclomatic": 5,
29            "cyclomaticDensity": 12.821,
30            "halstead": {
31              "bugs": 0.524,
32              "difficulty": 22.632,
33              "effort": 35598.646,
34              "length": 251,
35              "time": 1977.703,
36              "vocabulary": 77,
37              "volume": 1572.963,
38              "operands": {
39                "distinct": 57,
```

```

40         "total": 129,
41         "identifiers": [
42             "__stripped__"
43         ]
44     },
45     "operators": {
46         "distinct": 20,
47         "total": 122,
48         "identifiers": [
49             "__stripped__"
50         ]
51     }
52 },
53 "params": 6,
54 "sloc": {
55     "logical": 39,
56     "physical": 87
57 }
58 }
59 },
60 "module": "frontend/imperative/src/app.js",
61 "maintainability": 70.429
62 }
63 }
64 ]
65 }

```

F Imperative frontend application logic (vanilla JavaScript)

```
1  const senderEl = document.getElementById('sender');
2  const messageEl = document.getElementById('message');
3  const messageListEl = document.getElementById('chat');
4  const formEl = document.getElementById('form');
5  const messageWebsocket = new WebSocket('ws://localhost:8001');
6  const messages = [];
7  let me = '';
8
9  const debounce = (func, wait) => {
10     let timeout;
11     return () => {
12         let context = this, args = arguments;
13         const later = () => {
14             timeout = null;
15             func.apply(context, args);
16         };
17
18         clearTimeout(timeout);
19         timeout = setTimeout(later, wait);
20     };
21 };
22
23 const createSenderAvatar = (sender) => {
24     const avatarImg = document.createElement('img');
25     avatarImg.classList.add('chat__entry__avatar');
26     avatarImg.src = 'https://api.adorable.io/avatars/32/${sender}.svg';
27     avatarImg.title = sender;
28     avatarImg.alt = sender;
29
30     return avatarImg;
31 };
32
33 const createMessageSpan = (message) => {
34     const messageSpan = document.createElement('span');
35     messageSpan.classList.add('chat__entry__message');
36     const messageText = document.createTextNode(message);
37     messageSpan.appendChild(messageText);
38
39     return messageSpan;
40 };
41
```

```

42 const createMessage = (message) => {
43   const messageLi = document.createElement('li');
44   const currentSender = senderEl.value;
45   const isMine = currentSender === message.sender;
46   const classes = isMine ? ['chat__entry', 'chat__entry--mine'] :
      ['chat__entry'];
47   messageLi.classList.add(...classes);
48
49   if (!isMine) {
50     const avatarImg = createSenderAvatar(message.sender);
51     messageLi.appendChild(avatarImg);
52   }
53
54   const messageSpan = createMessageSpan(message.message);
55   messageLi.appendChild(messageSpan);
56   return messageLi;
57 };
58
59 const addMessage = (message) => {
60   const messageLi = createMessage(message);
61   messageListEl.appendChild(messageLi);
62 };
63
64 const onSubmitDebounce = debounce((event) => {
65   const message = messageEl.value;
66   const sender = senderEl.value;
67
68   if (message.trim().length > 0 && sender.trim().length > 0) {
69     const newMessage = { message, sender };
70     messageWebsocket.send(JSON.stringify(newMessage));
71   }
72 }, 200);
73
74 formEl.addEventListener('submit', onSubmitDebounce);
75 formEl.addEventListener('submit', (event) => {
76   event.preventDefault();
77 });
78
79 senderEl.addEventListener('input', (event) => {
80   const newSender = event.target.value;
81   const notMyMessagesIndexes = [];
82   const myMessagesIndexes = [];
83
84   if (newSender.trim() !== me) {
85     for (let i = 0; i < messages.length; i++) {
86       const message = messages[i];
87       if (message.sender === me) {

```



```

88     const messageLi = messageListEl.childNodes.item(i);
89     messageLi.classList.remove('chat__entry--mine');
90     messageLi.insertBefore(createSenderAvatar(message.sender),
        messageLi.firstChild);
91 } else if (message.sender === newSender) {
92     const messageLi = messageListEl.childNodes.item(i);
93     messageLi.classList.add('chat__entry--mine');
94     messageLi.removeChild(messageLi.firstChild);
95 }
96 }
97 }
98 me = newSender.trim();
99 });
100
101 messageWebsocket.addEventListener('message', (event) => {
102     const message = JSON.parse(event.data);
103     messages.push(message);
104     addMessage(message);
105 });

```

G Plato report for vanilla JavaScript implementation

```
1  {
2    "summary": {
3      "total": {
4        "eslint": 0,
5        "sloc": 106,
6        "maintainability": 69.146
7      },
8      "average": {
9        "sloc": 106,
10       "maintainability": "69.15",
11       "eslint": "0.00"
12     }
13   },
14   "reports": [
15     {
16       "info": {
17         "file": "src/app.js",
18         "fileShort": "src/app.js",
19         "fileSafe": "src_app_js",
20         "link": "files/src_app_js/index.html"
21       },
22       "eslint": {
23         "messages": 0
24       },
25       "complexity": {
26         "aggregate": {
27           "complexity": {
28             "cyclomatic": 9,
29             "cyclomaticDensity": 11.538,
30             "halstead": {
31               "bugs": 1.062,
32               "difficulty": 33.103,
33               "effort": 105486.821,
34               "length": 469,
35               "time": 5860.379,
36               "vocabulary": 111,
37               "volume": 3186.581,
38               "operands": {
39                 "distinct": 87,
40                 "total": 240,
41                 "identifiers": [
```

```
42         "__stripped__"
43     ]
44 },
45     "operators": {
46         "distinct": 24,
47         "total": 229,
48         "identifiers": [
49             "__stripped__"
50         ]
51     }
52 },
53     "params": 10,
54     "sloc": {
55         "logical": 78,
56         "physical": 106
57     }
58 }
59 },
60     "module": "src/app.js",
61     "maintainability": 69.146
62 }
63 }
64 ]
65 }
```
