UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Urmas Talimaa

# Reactive Visualizer: A Learning Tool for Reactive Programming Using Reactive Extensions for JavaScript

Bachelor's Thesis (6 ECTS)

Supervisor: Aivar Annamaa

Tartu 2015

## Reactive Visualizer: A Learning Tool for Reactive Programming Using Reactive Extensions for JavaScript

**Abstract:**
Reactive programming enables declarative composition of asynchronous processes, making asynchronous programs simpler to write and understand. The downside of reactive programming is that learning it requires considerable effort, which can overwhelm beginners. An interactive learning tool can help beginners comprehend reactive programming by visualizing asynchronous interactions, providing helpful examples and guaranteeing a valid program structure. This thesis describes the design and implementation of Reactive Visualizer, a tool which aims to fulfill those requirements.

**Keywords:** Reactive programming, Reactive Extensions for JavaScript, JavaScript, Programming, Computer Science, Education

## Reactive Visualizer: reaktiivse programmeerimise õppevahend teegile Reactive Extension for JavaScript

**Lühikokkuvõte:**
Reaktiivne programmeerimine võimaldab asünkroonsete protsesside ja nendevaheliste seoste deklaratiivset kirjeldamist, lihtsustades asünkroonsete programmide kirjutamist ja mõistmist. Reaktiivse programmeerimise õppimist takistab aga asjaolu, et selle õppimine on raske ja nõuab algajatel tugevat pingutust. Reaktiivse programmeerimise õppimist lihtsustaks interaktiivne õppevahend, mis visualiseerib asünkroonseid seoseid, pakub abistavaid näiteid ja garanteerib korrektse programmistruktuuri. Antud bakalaurusetöö kirjeldab õppevahendi Reactive Visualizer, mis püüab eelnimetatud nõudeid täita, disaini ja implementatsiooni.
**Võtmesõnad:** Reaktiivne programmeerimine; Reactive Extensions for JavaScript; JavaScript; Programmeerimine; Informaatika; Haridus

# Contents

# Introduction

Reactive programming is a programming paradigm which aims to declaratively describe values which change over time. The values are usually modified and combined using stateless functions, many of them similar to functions defined on lists in functional programming. Common uses of reactive programming include real-time systems, interactive web pages and games. A library allowing reactive programming in the web browser and JavaScript-based servers is Reactive Extensions for JavaScript [1] by Microsoft Open Technologies.

The goal of this thesis is to design and implement a web-based learning tool for reactive programming, called Reactive Visualizer. The author's experience in the private sector confirms the usefulness of reactive programming, however learning it has required considerable effort. This thesis aims to lessen that effort by creating an easily-accessible interactive learning tool.

Reactive Visualizer is based on the Reactive Extensions for JavaScript library, but aims to provide an introduction to reactive programming in general. It can be accessed from a standard web browser, requiring no installation of plug-ins. The user interface for Reactive Visualizer is written using the React [2] library by Facebook. All of the code in Reactive Visualizer is written in CoffeeScript [3] and the source code is made publicly available under the open source MIT license.

The thesis is divided into six chapters. The first chapter gives a brief overview of asynchronous programming and more specifically, reactive programming. The second chapter concentrates on the Reactive Extensions for JavaScript library and its application programming interface (API). Motivation and the requirements for Reactive Visualizer are given in the third chapter. The fourth chapter describes the user interface and the fifth the implementation details of Reactive Visualizer. The sixth chapter lists related work with comparisons to Reactive Visualizer.

The code listings in this thesis are given in CoffeeScript [3].

Reactive Visualizer can be accessed at https://urmastalimaa.github.io/reactive-visualizer/ with the source code being available at https://github.com/urmastalimaa/reactive-visualizer.

Diagrams describing complex RxJS concepts, the manual included with Reactive Visualizer and initial reviews are provided as appendices.

# 1 Background in Reactive Programming

## 1.1 The Asynchronous Programming Model

Rodrigo Fonseca [4] from Brown University gives an overview of the synchronous and asynchronous programming models. Given a program with three distinct tasks Fonseca describes how different programming models can be used to control the program flow: [4]

- A synchronous single-threaded program will execute the tasks one by one in the defined order, resulting in a very simple program.
- A multi-threaded program will try to execute all the tasks in parallel, giving the control of managing the threads to the operating system. The multi-threaded program can become very complex if the tasks need coordination.
- An asynchronous program is able to interleave the tasks while running in a single thread. The asynchronous program will be simpler than a multi-threaded one while still allowing tasks to be interleaved.

While an asynchronous program will be much more complex than a synchronous one, it can wait indefinitely for external output with using little to no resources. This makes the asynchronous programming model ideal for highly interactive programs like soft real-time systems or web applications.

## 1.2 Asynchronous Programming Using JavaScript

Programming for the web browser has changed vastly from the conception of the World Wide Web to the widespread use it sees today. Taivalsaari et al. [5] describe the evolution of the web from simple static pages, containing only text and images, to the rich pages containing plugins and JavaScript, to the final shift towards collaborative and interactive web applications.

One of the the first technologies towards an interactive web was Ajax. Garrett et al. [6] describe it as a programming model which incorporates numerous technologies to allow asynchronous user interaction with the web page. Allowing asynchronous interaction with a web page means that every user action can trigger a JavaScript method instead of a new page request. Some of the actions can be handled immediately in the browser, some might require an asynchronous XML request to the server, but both allow the user to continue interacting with the page without waiting for a response. [6]

In addition to Ajax, complementary technologies allowing real-time applications have emerged. Fette and Melnikov [7] published the WebSocket protocol which allows two-way communication between a client and a server. Bergkvist et al. [8] released a real-time peer-to-peer communication protocol in the browser called WebRTC. Both of these technologies rely heavily on the the asynchronous programming model provided by JavaScript's Event Loop [9].

The widespread use of the World Wide Web has created an influx of JavaScript programmers which in turn has resulted in JavaScript escaping the browser and finding its way onto web servers (e.g. Node.js [10]) and mobile devices (e.g. React Native [11]). Thus the need for tools that allow writing easy to understand and scalable JavaScript is greater than ever.

## 1.3 A Different Source of Complexity

A popular way of reducing program complexity is to use functional programming. John Hughes [12] describes functional programs as programs containing no assignment statements nor side effects. Hughes adds that the removal of side effects removes a major source of bugs and helps to decompose and modularise programs [12].

While the functional programming style is highly useful in the synchronous parts of a program, it does not help with reducing the complexity produced by asynchronous flow control. When an user input can cause concurrent requests to different web servers, each requiring different strategies for retrial and cancellation, with the results of all the requests requiring further combination, the flow of a program can be very hard to understand. From the author's experience the failure to describe and understand the asynchronous dependencies are the biggest sources of errors in highly interactive programs.

## 1.4 The Problem With Callbacks

There are two popular techniques for handling asynchronous dependencies:

- Promises [13] can be used to describe asynchronous computations (e.g. web requests),
- EventListeners [14] can be used to register listeners on the user interface elements of a web page or any object that supports events.

Both of the approaches use a simple callback mechanism for notifying listeners and offer little to no composition options. Although Promises support a few composition functions [13], EventListeners support none and there is no built-in interoperability between EventListeners and Promises. The callback mechanism used by EventListeners is demonstrated in listing 1.

```
document.getElementById('save').addEventListener(
  'click', # the event that we are interested in
  (event) -> saveWork() # the callback which will be called on a click event
)
```

Listing 1: Listening for click events

The easiest way to compose asynchronous processes is to manually compose them in the callbacks. Listing 2 shows an example of such composition using an EventListener (click) and a Promise (web request).

Syme, Petricek, and Lomov [15] propose that normal asynchronous programming produces a tangle of callbacks, exceptions and data races. They describe callbacks as being awkward, unexpressive and introducing unnecessary inversion of control [15]. Indeed, this is already the case with the code in listing 2 as two consecutive clicks on different items create a data race in which the result for the first click might arrive after the result for the second click. Even more, if the request for the first item times out then a perfectly good response for the second item might be overwritten with an error.

```
itemQueryElements = document.getElementsByClassName('itemQuery')
resultElement = document.getElementById('itemQueryResult')

for itemQueryElement in itemQueryElements

  itemQueryElement.addEventListener('click', (event) ->
    itemId = event.target.attributes.itemId.value
    # The user requested information about an item with id `itemId`.
    # Request information from the web server
    #   and display the result in the `itemQueryResult` element.
    itemPromise = requestItemFromServer(itemId)

    itemPromise.then(
      (result) -> displayResult(result, resultElement)
      (error)  -> displayError(error, resultElement)
    )
  )
```

Listing 2: Manual callback composition of a click event and a web request

## 1.5 Reactive Programming

Wikipedia defines reactive programming as "a programming paradigm oriented around data flows and the propagation of change." [16]. A more formal definition is given by Elliott [17] describing functional reactive programming as composition of time-varying values - behaviors and streams of timed values - events using the Haskell programming language.

A simpler definition which is more fitting for JavaScript is given by Liberty, Betts, and Turalski [18]. They propose thinking about event sources as lists of asynchronous information and describe functional reactive programming as combining these asynchronous lists using functional programming [18, p. xvi]. The Reactive Extensions (Rx) library and its JavaScript port Reactive Extensions for JavaScript (RxJS) provide such asynchronous lists and the operators to combine them. It is the RxJS library that this thesis is mainly based upon.

# 2 The Reactive Extensions for JavaScript

A comprehensive description of the Reactive Extensions for JavaScript library is outside of the scope of this thesis, but a general overview of the main abstractions is given as they are used greatly in chapters 3, 4 and 5. "Programming Reactive Extensions and LINQ"[18] and the various resources at the resources section on the RxJS project repository page [1] can provide a deeper understanding of Reactive Extensions.

The RxJS repository page describes Reactive Extensions for JavaScript as "a set of libraries to compose asynchronous and event-based programs using observable collections and Array#extras style composition in JavaScript"[1]. Array#extras provide standardized methods for processing *array objects* in JavaScript, including higher-order methods that are usually associated with functional programming [19]. An observable collection is a specification of a push-based collection, which will be described in the next paragraph.

A traditional pull-based collection is passive, requiring the user of the collection to actively query for items, check for the completion of the collection and handle exceptions. A pull-based collection is unsuitable to describe an asynchronous information source, as it is unknown if and how many items the source will produce. On the other hand a push-based collection does not provide a querying mechanism, instead notifying the user of the collection of new items, errors and completion similarly to the observer pattern. This makes the push-based collection a better fit for describing an asynchronous information source.

## 2.1 The Observable and the Observer

The core abstraction of Reactive Extensions is the Observable object. The Observable object defines the properties of the push-based asynchronous collections that Reactive Extensions is based on.

As a push-based collection the Observable object supports only one method for retrieving items: `subscribe`. The `subscribe` method accepts as its only parameter an Observer object.

The Observer interface defines three methods: `onNext`, `onError` and `onCompleted`. Note that in RxJS an Observer is often provided not as a separate object, but as three functions. When a new item is made available in an observable collection, the `onNext` callback is called with the item, when an error occurs the `onError` callback is called with the error and when no more items are expected the `onCompleted` callback is invoked. [18, p41-42]

What makes subscribing to an Observable using callbacks different from simply using Promises or EventListeners, is that many Observable objects can be composed to create the required final Observable. This postpones the use of the callback mechanism to the stage when no further composition is necessary. Listing 3 shows how the item querying code from listing 2 can be rewritten using Observables, fixing the data race in the process. The functions used in listing 3 are described in sections 2.2 and 2.3.

```
itemQueryElements = document.getElementsByClassName('itemQuery')
resultElement = document.getElementById('itemQueryResult')

for itemQueryElement in itemQueryElements

  itemQuery = Rx.Observable.fromEvent(itemQueryElement, 'click')
    .map((event) -> event.target.attributes.itemId.value)

  itemQueryResult = itemQuery.flatMapLatest((itemId) ->
    Rx.Observable.fromPromise(requestItemFromServer(itemId))
  )

  itemQueryResult.subscribe(
    (result) -> displayResult(result, resultElement)
    (error)  -> displayError(error, resultElement)
  )
```

Listing 3: Composition of a click event and a web request using Observables

The RxJS grammar that is given in listing 4 defines how the Observer callbacks are invoked. The `onNext` callback can be called zero or more times, optionally followed by a single invocation of either the `onCompleted` or the `onError` callback. The grammar allows consumers of the Observable to perform cleanup after either the `onError` or the `onCompleted` callback is invoked as it is guaranteed that no more items are produced. [20]

```
onNext* (onCompleted | onError)?
```

Listing 4: The RxJS grammar

The Observable and Observer define the characteristics of the asynchronous collection but are of little value on their own. Sections 2.2 and 2.3 describe the factories and operators which allow the creation and composition of observable collections.

## 2.2 Factories

The Observable object defines multiple factory methods to provide the initial values for an observable sequence: [21]

- Factories which lift objects (`.just`) or synchronous collections (`.fromArray`, `.range`, `.repeat`) into observable collections;
- Factories which transition other sources of asynchronous behaviour into observable collections (`.fromEvent`, `.fromPromise`, `.fromCallback`);
- Factories which create asynchronous values (`.timer`, `.interval`);
- A factory which creates an observable containing an error (`.throw`).

Every Observable is created using an initial factory method and the Observables can be further composed using operators. Listing 6 provides example usage of some of the previously mentioned factory methods.

Listings 6, 7 and 9 use a simple logging observer, which is given in listing 5, and describe the expected results as lines of comments following the Observable.

```
createObserver = ->
  Rx.Observer.create(
    (item) -> console.log(item)
    (error) -> console.log("error:", error, "\n")
    -> console.log("completed\n")
  )
```

Listing 5: A function for creating a logging Observer

```
Rx.Observable.just(5).subscribe(createObserver())
# 5
# completed

Rx.Observable.fromArray([1,2,3]).subscribe(createObserver())
# 1
# 2
# 3
# completed

Rx.Observable.range(1, 3).subscribe(createObserver())
# 1
# 2
# 3
# completed

emitter = new EventEmitter
Rx.Observable.fromEvent(emitter, 'event2').subscribe(createObserver())

emitter.emit('event1', 1)
emitter.emit('event2', 2)
# 2

Rx.Observable.throw(new Error("an error has occured")).subscribe(createObserver())
# error: [Error: an error has occured]

Rx.Observable.timer(500).subscribe(createObserver())
# 0 (after 500 ms)
# completed (after 500 ms)
```

Listing 6: Examples of factory methods

## 2.3 Operators

The operators included in RxJS provide the main benefit of the library: the composition of asynchronous computations. As the composition of asynchronous computations can be highly complex, RxJS provides many different operators. The Observable operators can be loosely separated into five categories (based on RxJS categorization of operators [22]):

- Operators for selecting and filtering values;

- Operators for selecting and filtering values using another Observable;
- Operators for grouping values;
- Exception handling operators;
- Time-based operators.

A comprehensive overview of operators and examples of their usage can be found at ReactiveX [23].

### Operators for Selecting and Filtering Values

This category contains operators which are also applicable to synchronous lists, which makes the operators relatively easy to understand. Some of the operators in this category are: [21]

- `map` - projects every element into a new form;
- `filter` - filters the elements based on a predicate;
- `find` - finds the first occurrence of an element matching a predicate;
- `take` - returns the specified number of elements;
- `skip` - skips the specified number of elements.

Listing 7 illustrates the usage of these operators.

```
Rx.Observable.range(1,3)
  .map((x) -> x * x)
  .subscribe(createObserver())
# 1
# 4
# 9
# completed

Rx.Observable.range(1,3)
  .filter((x) -> x > 2)
  .subscribe(createObserver())
# 3
# completed

Rx.Observable.range(1,3)
  .find((x) -> (x % 2) == 0)
  .subscribe(createObserver())
# 2
# completed

Rx.Observable.range(1,3)
  .take(1)
  .subscribe(createObserver())
# 1
# completed

Rx.Observable.range(1,3)
  .skip(1)
  .subscribe(createObserver())
# 2
# 3
# completed
```

Listing 7: Examples of operators for selecting and filtering values

### Operators for Selecting and Filtering Values Using Other Observables

The operators in this category offer similar functionality to operators for selecting and filtering values but are highly more complex because they use another Observable and its values to perform the transformation.

Liberty, Betts, and Turalski [18, p xvi] warn the readers of "Programming Reactive Extensions and LINQ" with

> Rx will stretch your brain in ways it's not used to stretching, and you, like the authors, will almost certainly hit a learning curve.

In the opinion of the author of this thesis, the operators which use another Observable to perform a transformation make up most of that learning curve. Although these operators are hard to understand initially, they do allow the composition of different Observables, which makes them also the most useful.

A standard operation defined on synchronous lists is the flattening of a list of lists, which is described in listing 8.

```
listOfLists = [
  [1, 2, 3, 4, 5],
  [1, 4, 9],
  [1, 8, 27],
]
console.log(flatten(listOfLists))
# [1, 2, 3, 4, 5, 1, 4, 9, 1, 8, 27]
```

Listing 8: Flattening a synchronous list of lists

In contrast to the relative simple concept of flattening synchronous lists, Rx offers three different ways of flattening an Observable of Observables:

- `concatAll` - receives elements from the first Observable until it completes and only then starts receiving elements from the next Observable;
- `mergeAll` - receives elements from all the Observables simultaneously;
- `switch` - receives elements only from the latest produced Observable.

Figures 11, 12 and 13 in appendix A provide marble diagrams for the `concatAll`, `mergeAll` and `switch` operators. A marble diagram represents time flowing from left to right as a line, containing the values of the Observable on the line with the completion of the Observable represented by a vertical line [24].

Due to the different ways of flattening an Observable of Observables there are also corresponding operators for flat-mapping (mapping using a function which returns an Observable and flattening): [21]

- `concatMap` - (`map + concatAll`) concatenates the Observables in the order that they are produced;
- `flatMap` - (`map + mergeAll`) merges all the produced Observables;
- `flatMapLatest` - (`map + switch`) receives elements from the last produced Observable.

Figures 14, 15 and 16 in appendix A provide marble diagrams for the `concatMap`, `flatMap` and `flatMapLatest` operators.

In addition to the flat-mapping operators, some other operators in this category are: [21]

- `merge` - merges the argument Observable with the source Observable;
- `concat` - concatenates the argument Observable after the source Observable;
- `zip` - merges the argument Observable with the source Observable, using a selector function to combine the values when both Observables have produced a value;
- `combineLatest` - merges the argument Observable with the source Observable, using a selector function to combine the values when either of the Observables produces a value;

- `takeUntil` - returns values from the source Observable until the argument Observable produces a value;

Listing 9 illustrates the usage of the listed operators.

```
Rx.Observable.interval(300)
  .merge(Rx.Observable.interval(200))
  .take(6)
  .subscribe(createObserver())
# 0 (at 200 ms)
# 0 (at 300 ms)
# 1 (at 400 ms)
# 1 (at 600 ms)
# 2 (at 600 ms)
# 3 (at 800 ms)
# completed (at 800 ms)

Rx.Observable.timer(500)
  .concat(Rx.Observable.timer(200))
  .subscribe(createObserver())
# 0 (at 500 ms)
# 0 (at 700 ms)
# completed (at 700 ms)

Rx.Observable.interval(500)
  .zip(Rx.Observable.interval(200), ((a, b) -> {a: a, b: b}))
  .take(3)
  .subscribe(createObserver())
# { a: 0, b: 0 } (at 500 ms)
# { a: 1, b: 1 } (at 1000 ms)
# { a: 2, b: 2 } (at 1500 ms)
# completed (at 1500 ms)

Rx.Observable.interval(500)
  .combineLatest(Rx.Observable.interval(200), ((a, b) -> {a: a, b: b}))
  .take(5)
  .subscribe(createObserver())
# { a: 0, b: 1 } (at 500 ms)
# { a: 0, b: 2 } (at 600 ms)
# { a: 0, b: 3 } (at 800 ms)
# { a: 0, b: 4 } (at 1000 ms)
# { a: 1, b: 4 } (at 1000 ms)
# completed (at 1000 ms)

Rx.Observable.interval(500)
  .takeUntil(Rx.Observable.timer(2000))
  .subscribe(createObserver())
# 0 (at 500 ms)
# 1 (at 1000 ms)
# 2 (at 1500 ms)
# completed (at 2000 ms)
```

Listing 9: Examples of operators for selecting or filtering values using other Observables

#### Other Operators

RxJS also provides operators for grouping the elements of an Observable, which behave similarly to grouping operators which work on synchronous lists, with the addition of the time dimension: [21]

- `bufferWithCount` - projects each element into a buffer with the specified size;
- `bufferWithTime` - projects elements into buffers which are created based on the specified timespan;
- `groupBy` - groups the elements based on a key selector function and comparison function.

The RxJS grammar in listing 4 defined that an Observable can terminate either normally (`onCompleted`) or with an error (`onError`). The operators in the *Exception handling operators* category allow describing error cases: [21]

- `catch` - continues the Observable with the argument Observable if the source Observable terminates with an error;
- `retry` - repeats the source Observable the specified number times or until it terminates normally;
- `onErrorResumeNext` - continues the Observable with the argument Observable however the source Observable terminates.

As the dimension of time is of major importance to asynchronous lists, many of the operators in RxJS deal with time-based transformations: [21]

- `debounce` - emits the last item when the source Observable has not produced any new items for the specified timespan;
- `sample` - emits the latest item produced by the source Observable at each specified interval;
- `delay` - delays every item by the specified amount;
- `timeout` - throws an error if the source Observable does not produce any values for the specified time.

For further information and usage examples of RxJS operators refer to the documentation that is available at ReactiveX [23].

## 2.4 Schedulers

Operators which need to handle concurrency take an additional parameter: a Scheduler. The Scheduler specifies the concurrency context of the operator. [18, p89]

Every implementation of Reactive Extension provides different Schedulers based on the characteristics of the language. As JavaScript is a single-threaded language, there are only a few schedulers provided by RxJS: [25]

- `Rx.Scheduler.currentThread` schedules work as soon as possible, but does not support scheduling to a time in the future;
- `Rx.Scheduler.immediate` schedules work immediately;
- `Rx.Scheduler.default` schedules work via a platform specific timed callback (usually using window.setTimeout in the browser);
- `Rx.TestScheduler` schedules work using virtual time.

Schedulers can be used for fine-tuning the behaviour of some operators. Listing 10 shows the usage of the `generate` factory method, which produces ten million values. When the `Rx.Scheduler.currentThread` or the `Rx.Scheduler.default` Schedulers are used, the generator would generate a single value and schedule the next iteration to be processed later, allowing the `take` operator to act, which would notify the generator that no further values are needed. However when the `Rx.Scheduler.immediate` Scheduler is used, all ten million values would be generated in memory before the `take` operator would be able to notify the generator that only a single value is required.

```
Rx.Observable.generate(
  0,
  (x) -> x < 10000000
  (x) -> x + 1
  (x) -> x
  Rx.Scheduler.currentThread
 ).take(1)
  .subscribe(createObserver())
```

Listing 10: Example of fine-tuning the `generate` factory method using a Scheduler

Almost all of the operators which deal with concurrency use the `Rx.Scheduler.default` scheduler. Rx.TestScheduler is very useful for testing as it allows scheduling otherwise asynchronous operations synchronously, recording the timestamps that the operations were supposed to occur.

# 3 Requirements for Reactive Visualizer

## 3.1 Scope

Experience has shown that the steepness of RxJS' learning curve comes mainly from not knowing where to start, which operators to use and from misunderstanding the interactions between operators. To tackle these problems, a beginner-friendly learning tool is needed that can be used to explore reactive programming and RxJS. The tool should also be usable for learning and teaching the various operators that RxJS has to offer. Implementing the learning tool as a web application would make it accessible to all potential users.

The simplest way to debug an Observable or to learn an operator is to create an Observable in a JavaScript console and see the values that it produces. This has the drawback of not seeing the values of all the operators, which can be mitigated by adding logging statements to every operator. However, when changing the parameters of one operator the whole Observable must be re-evaluated and it is hard to see interactions between operators at a certain point in time. This is the problem that Reactive Visualizer aims to solve.

The goal of Reactive Visualizer is not to provide a debugger for RxJS nor is it supposed to visualize the behaviour of an actual program using RxJS. Chapter 6 provides information about tools which provide such functionality.

## 3.2 Functional Requirements

Molich and Nielsen [26] described the nine principles of good program usability:

- Simple and Natural Dialogue;
- Speak the User's Language;
- Minimize the User's Memory Load;
- Be Consistent;
- Provide Feedback;
- Provide Clearly Marked Exits;
- Provide Shortcuts;
- Provide Good Error Messages;
- Error Prevention.

These usability principles in conjunction with the goals of Reactive Visualizer help to derive the functional requirements:

- Provide a structured editor that provides default values and encourages a correct Observable structure;

- Visualize values moving through the operators;
- Allow inspecting the Observable at a certain point in time;
- Instantly and automatically update the visualized values when the Observable is changed in the editor;
- Provide links to the official documentation of operators;
- Output the Observable from the editor in a serializable format;
- Allow loading and saving named Observables;
- Constantly save the observable in the editor to avoid losing the user's work;
- Include an in-browser manual.

Fulfilling these requirements will satisfy the goals specified in section 3.1 in an user-friendly way.

## 3.3 Non-Functional Requirements

Robert Martin quotes Ron Jeffries in Martin's book "Clean code: a handbook of agile software craftsmanship" [27] on what Jeffries thinks that is *clean code*:

> In recent years I begin, and nearly end, with Beck's rules of simple code. In priority order, simple code:
>
> - Runs all the tests;
> - Contains no duplication;
> - Expresses all the design ideas that are in the system;
> - Minimizes the number of entities such as classes, methods, functions, and the like.

Deriving from Jeffries's requirements for clean code and the functional requirements, the following non-functional requirements were set for Reactive Visualizer:

- Small, testable modules;
- Immutable, testable components in the user interface;
- Functional programming style;
- Clear separation of the editor and the visualization modules;
- Synchronous evaluation of the Observable and the values moving through its operators.

Marchenko, Abrahamsson, and Ihme [28] found that using test-driven development (TDD) had perceived positive effects of improved code quality, readability and extensibility. Therefore Reactive Visualizer should be written using TDD, resulting in tested modules and user interface components and increased longevity of the project. Tested code also simplifies refactoring as CoffeeScript is a dynamically typed language in which refactoring is more difficult compared to static languages [29].

Immutable user interface components will simplify reusing the components, which simplifies modifying Reactive Visualizer for future requirements.

Using functional programming helps to reduce the amount of bugs and helps to modularise the program [12].

Separating the simulation and editor will allow creating an editor for a different language by switching out the editor or creating an editor for a different reactive programming library by switching out the simulation module. A list of languages which compile to JavaScript (and can therefore be used in the browser) can be found at the CoffeeScript language wiki page [30]. Kambona, Boix, and De Meuter [31] provide a list of reactive programming libraries which are available in the web browser.

In order to instantly change the simulation values when the Observable changes, the Observable must be synchronously evaluated by simulating asynchronous behavior using virtual time.

# 4 User Interface of Reactive Visualizer

Reactive Visualizer is designed to run in the user's web browser. The application logic is loaded into the web browser and no other interaction with a web server is necessary.

## 4.1 React

The user interface components of Reactive Visualizer are built using the React [2] library. React enables writing immutable, reusable user interface components [32]. React components are immutable by default, depending solely on their input properties (`props`) and optional internal state (`state`).

Whenever the properties or state of a React component change, it and all of its child-components are re-rendered. Re-rendering keeps the components up-to-date and makes the data-flow through them easy to understand. In addition it allows Reactive Visualizer to easily load and save the Observable by simply changing the input properties of all components whenever the Observable changes.

## 4.2 User Interface Elements

There are five distinct user interface elements as can be seen in figure 1:

- Manual,
- Observable editor,
- Virtual time controller,
- Inspector,
- Persistence.

The user interface elements map directly to React components with the same name.

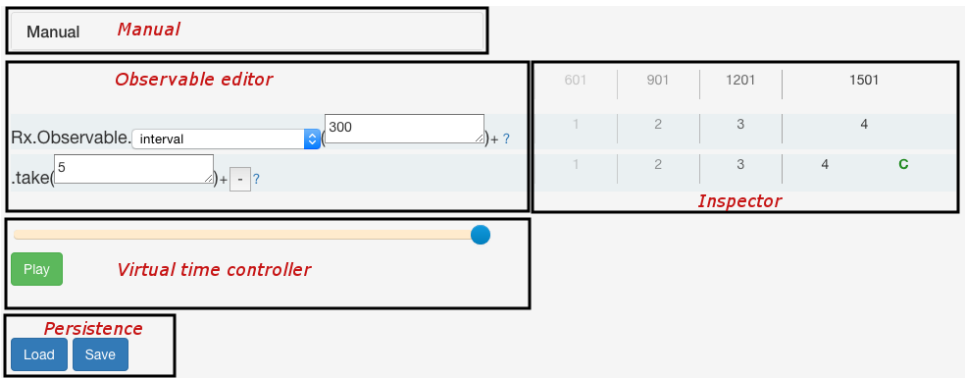


Figure 1: User interface

## 4.3 Manual

The manual component is designed to introduce Reactive Visualizer and give an overview of the user interface components and describe how to use them. It is divided by the main user interface parts as can be seen in figure 2. The full manual is included in appendix B.

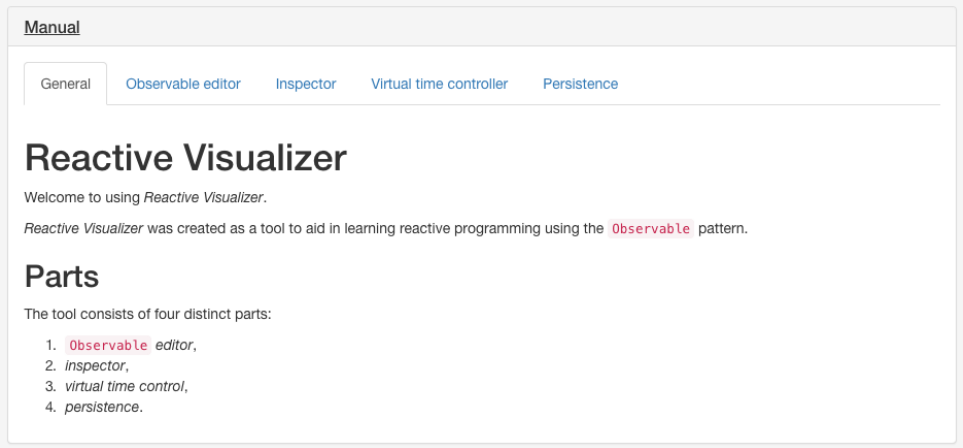


Figure 2: Manual component

## 4.4 Observable Editor

The Observable editor is a code editor for creating an Observable. The editor is designed so that different Observables can be easily created even if the user is unfamiliar with RxJS.

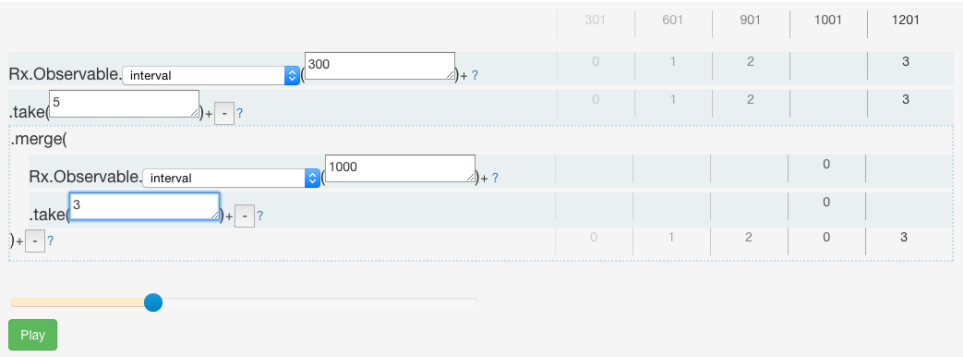


Figure 3: An operator with Observable as an argument

### 4.4.1 Factories

An Observable is created using a single Observable factory which can be selected from a drop-down menu. The drop-down is prefixed with Rx.Observable. to display syntactically correct code.

The arguments for the Observable factory (if any) can be provided in automatically created text inputs. A separate input will be created for every argument and a default value is always provided.

After every Observable factory there is a ? symbol which opens the official documentation page for that factory Observable.

### 4.4.2 Operators

The factory can be followed by any number of of operators.

Every factory and operator is followed by a + symbol on its right side. An operator can be added by selecting an operator from the drop-down which appears when hovering over the + symbol.

Similarly to Observable factories, a text input will be generated and populated with a default value for every expected argument of the operator.

Every operator is followed by a – symbol which removes the operator.

On the far right side of the operator there is a ? symbol which opens the official documentation page for the operator.

### 4.4.3 Arguments Expecting Observables

Although the Observable editor outputs only a single Observable there can be many different Observables combined using various operators. One such operator, merge, which combines the values from the source and argument Observable, can be seen on figure 3.

An argument which expects an Observable is populated with another Observable editor. In the author's opinion, operators expecting other Observables are the hardest to understand. With Reactive Visualizer the user can easily identify such operators and use an already familiar editor to provide the argument Observable.

## 4.5 Virtual Time Controller

The Virtual time controller (displayed on figure 4) allows setting the internal clock of the Observable.

There are two ways to change the internal clock of the Observable.

- Use a slider to change the time to any point in the lifetime of the Observable.
- Replay the Observable in real time using the stored timing data. The playback starts from the previously set virtual time and continues until a new time is selected or until the Observable completes.

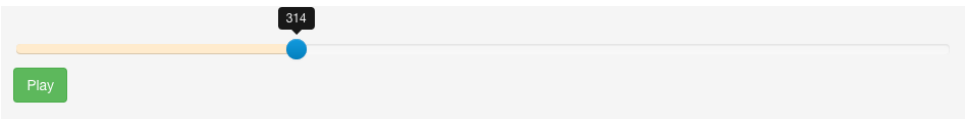


Figure 4: The Virtual time controller

## 4.6 Inspector

The Inspector shows the values produced by the virtual time controller.

The RxJS contract specifies three kinds of callbacks for an Observer: `onNext`, `onError` and `onCompleted`. As can be seen in figure 5, the Inspector will display:

- an object from the `onNext` callback as a string,
- an error from the `onError` callback as red-coloured error message,
- a completion from the `onCompleted` callback as a green `C` symbol.

The Inspector will show only values which have occurred up to the time specified in the virtual time controller. A new column will be created in the Inspector whenever any sub-Observable produces a value.

In order to show the user how previous values can have an effect on future values, some values from the past are also shown. As new values appear, past values will be shifted to the left and their opacity will be reduced. This creates a virtual time-line next to the observable factory or operator, describing its characteristics.

The Inspector columns and the associated Observable editor rows share a faintly colored background. This allows the user to easily distinguish which values in the Inspector originate from which Observable operator.

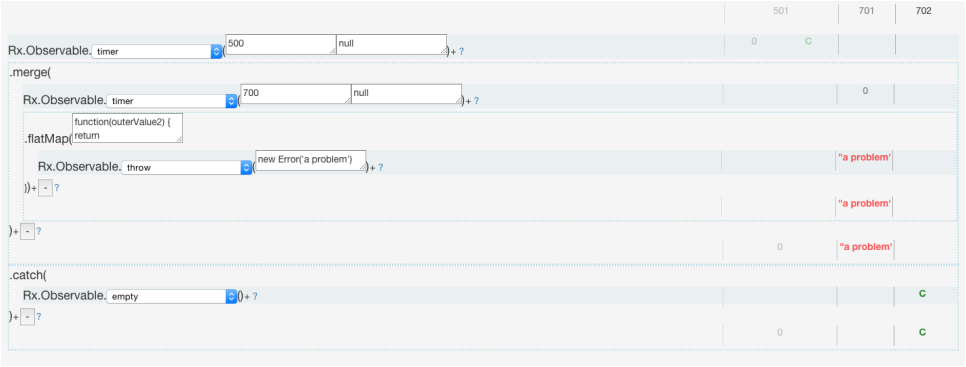


Figure 5: Inspector displaying values from `onNext`, `onError` and `onCompleted`

## 4.7 Persistence

The persistence component fulfills the requirement of saving and loading an Observable.

### 4.7.1 Loading an Example

The persistence component allows loading of example Observables via the *Load* button. Clicking on the *Load* button opens a modal (displayed on figure 6) where examples categories are listed.

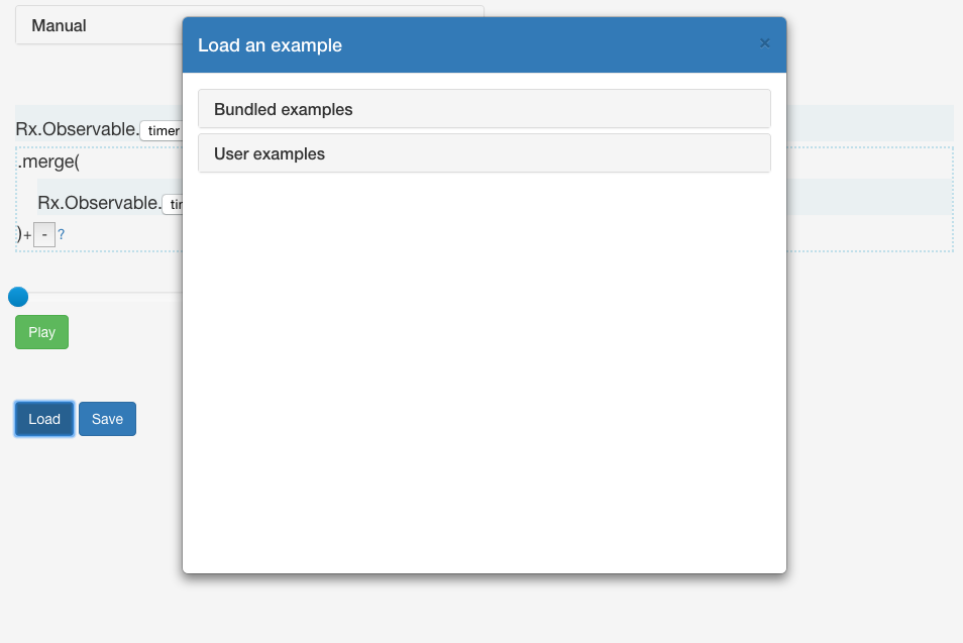


Figure 6: Loading an example

There are two categories of examples:

- Bundled examples, which can be seen on figure 7, cannot be changed by the user. They provide example observables which illustrate how RxJS can be used to solve basic asynchronous problems.
- User defined examples, which can be seen on figure 8, allow the user to save different observables which they might want to revisit later. This is similar to the *Save/Load* functionality provided by native applications. In contrast to the bundled examples, user defined examples can be removed.

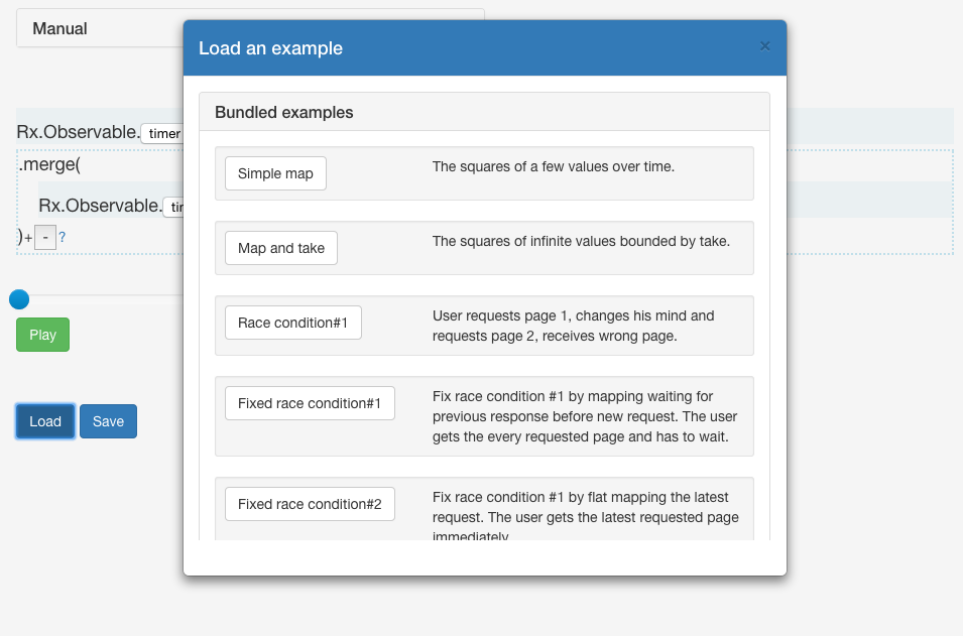


Figure 7: Bundled examples

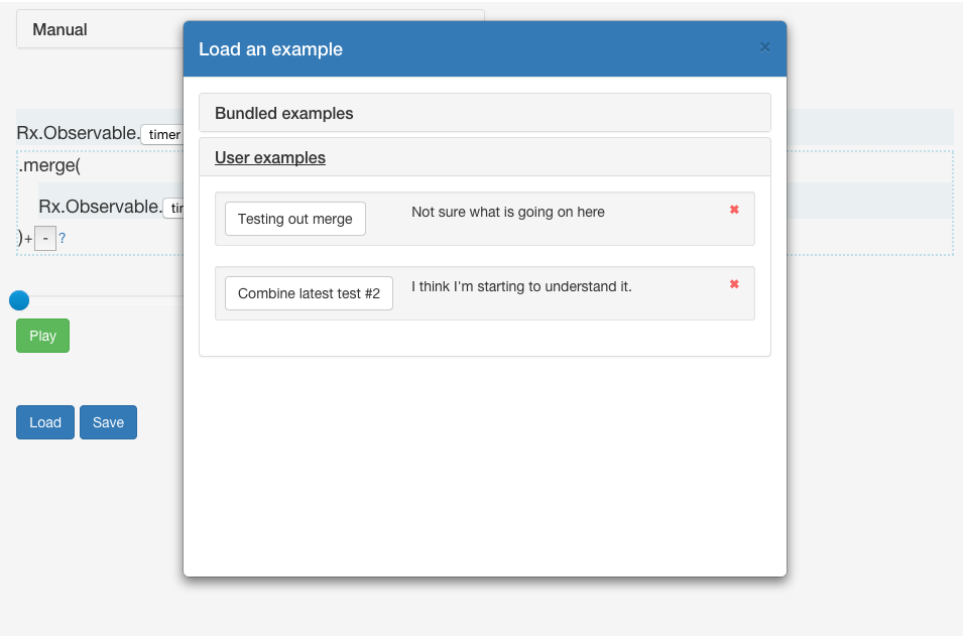


Figure 8: User defined examples

### 4.7.2 Saving an Example

The persistence component allows saving example observables via the *Save* button.

The saving functionality complements user defined examples. Clicking on the *Save* button opens a model with two mandatory inputs: name and description. The modal

can be seen on figure 9. After saving, an example with the provided name and description will be shown in user defined examples.

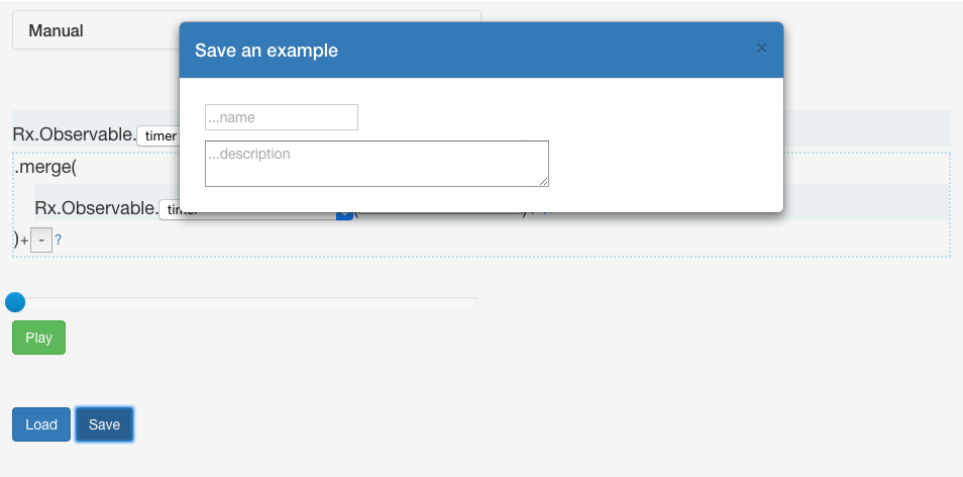


Figure 9: Saving an example

# 5 Implementation of Reactive Visualizer

## 5.1 Architecture Overview

The user interface components and application logic modules are cleanly separated and their inputs and outputs well-defined. Figure 10 shows the dependencies between modules and the state and properties of the React components. On the file level, Browserify [33] is used the modularize the CoffeeScript code.

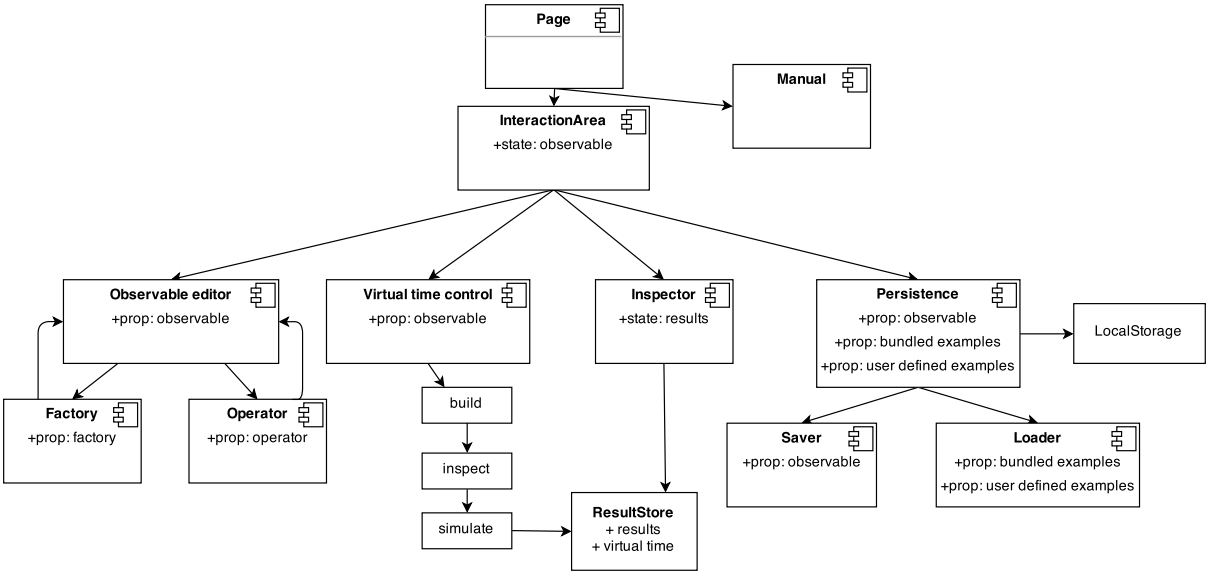


Figure 10: Dependency graph

## 5.2 Supporting Libraries

In order to fulfill the requirements set for Reactive Visualizer, two main supporting libraries are used.

**RxJS** RxJS [1], the reactive programming library on which Reactive Visualizer is based on, is used for evaluating the Observable that is created in the editor. Its testing modules are used for simulating the Observable using virtual time.

**Ramda** The developers of Ramda describe it as "A practical functional library for Javascript programmers."[34]. Ramda is used throughout Reactive Visualizer for writing concise and easily understandable functions. Ramda enables function currying and composition, which are not natively present in JavaScript. It also provides many useful functions for data manipulation.

## 5.3 Dependencies Between User Interface Components

The React user interface components usually communicate only via properties, making their dependencies well defined. The most used property is the observable description (named simply `observable` in figure 10). Whenever a component changes the Observable, the change is propagated upwards to the InteractionArea component which updates its `state observable` and propagates the change to all of its child components.

An exception is the dependency between the Virtual time controller and the Inspector. The Virtual time controller produces the results of the Observable simulation and the desired virtual time. However the Inspector is the only component that depends on those results, making it wasteful to propagate the change up to InteractionArea and re-render all the components. Therefore they communicate via a ResultStore, which notifies the Inspector whenever the time or the results change.

## 5.4 Identifying the Operators

Section 4.6 stated that the Inspector must show captured values for every Observable operator. Therefore during the evaluation of the Observable the captured values must be correlated with their origin. The easiest solution is to assign every operator an unique id which the Inspector can use to store the captured values. The id is derived from the position of the operator in the Observable chain.

## 5.5 Output of the Observable Editor

As stated in section 3.2, the Observable must be represented in a serializable way in the editor. Therefore it is necessary to specify the structure of the Observable representation. The representation (as shown in listing 11) contains for every Observable the factory method that is used to create it and a list of all the operators that are applied to the Observable. Every factory and operator also have a unique id (see section 5.4) associated with it. If an operator has another Observable as its argument, the same representation is used for the argument Observable.

```
{
  observable: {
    factory: {
      id: "f",
      type: "timer",
      args: [ 1000 ]
    },
    operators: [
      {
        id: "fo",
        type: "merge",
        args: [
          {
            observable: {
              factory: {
                id: "fo0f",
                type: "timer",
                args: [500]
              },
              operators: []
            }
          }
        ]
      },
      {
        id: "foo",
        type: "delay",
        args: [ 300 ]
      }
    ]
  }
}
```

Listing 11: Example output from the Observable editor

## 5.6 Simulating the Observable Using Virtual Time

The Observable is simulated using the Builder, Inspection wrapper and Simulator modules.

### 5.6.1 Builder

The Builder uses the editor output to create an Observable object. The editor output is interpreted and an Observable object is created with the correct factories, operators and arguments. Listing 12 shows the Observable which will be produced from the description in listing 11.

```
Rx.Observable.timer(1000)
    .merge(
      Rx.Observable.timer(500)
    )
    .delay(300)
```

Listing 12: The Observable produced by the Builder from the description in listing 11

### 5.6.2 Inspection wrapper

The Inspection wrapper is responsible for storing the values that flow through every factory and operator during the simulation.

The Builder calls the Inspection wrapper during the creation of the Observable. The Inspection wrapper appends a `do` operator after every operator. The `do` operator acts as an Observer, its callbacks will be called when a value, error or completion flows through the operator, but the operator supports no way of modifying the values. It is usually used to separate side-effects (e.g. debugging, logging) from other operators [21].

The Inspection wrapper uses the `do` operator to store the values moving through the operators. Listing 13 shows a simplification of the Observable produced by the Builder in conjunction with the Inspection wrapper.

```
createStoringObserver = (id) ->
  Rx.Observer.create(
    ((value) -> storeValue(id, value)),
    ((error) -> storeError(id, error)),
    (-> storeCompletion(id))
  )

Rx.Observable.timer(1000)
    .do(createStoringObserver('f'))
    .merge(
      Rx.Observable.timer(500)
        .do(createStoringObserver('foOf')
    )
    .do(createStoringObserver('fo'))
    .delay(300)
    .do(createStoringObserver('foo'))
```

Listing 13: The Observable produced by the Builder and Inspection wrapper from the description in listing 11

### 5.6.3 Simulator

The Simulator is responsible for simulating the Observable using virtual time.

An integral part of the Simulator is the TestScheduler [35], which is provided by the RxJS testing utilities. The TestScheduler is able to schedule any asynchronous operator synchronously using virtual time. However when using the TestScheduler

it is necessary to provide that scheduler to every asynchronous operator (otherwise `Rx.Scheduler.default` would be used). Due to this the Builder does not actually create an Observable object but rather provides a function which takes a Scheduler as an argument.

## 5.7 Persistence

In contrast to native applications a web application does not have many options for persistence. A simple and widely supported option is LocalStorage. LocalStorage allows persisting strings to the Storage object associated with the web page domain [36]. Fortunately, the Observable description is easily converted to a string as was demonstrated in section 5.5.

### 5.7.1 Saving the Current Observable

As the Observable editor outputs a new Observable description whenever changes happen in the editor, the persistence component will also receive a new Observable description immediately upon changes.

React provides a simple callback hook named `componentWillReceiveProps` which will be called whenever the properties of the component change. Listings 14 and 15 show how the current Observable, that the user is working on, is saved using `componentWillReceiveProps` and `localStorage`.

```
Persistence = React.createClass
    componentWillReceiveProps: (props) ->
        # Save the observable immediately when changed
        Persister.save(props.observable)
```

Listing 14: Persistence component reacting to the observable property change

```
Persister =
    saveObservable: (observable) ->
        localStorage.savedObservable = JSON.stringify(observable)
```

Listing 15: Persister module saving the observable using localStorage

## 5.8 Current Status

All initially planned functionality for Reactive Visualizer has been implemented and most of the non-functional requirements are fulfilled. The application is stable and the code-base is tested. Reactive Visualizer is currently deployed as a GitHub application at https://urmastalimaa.github.io/reactive-visualizer/.

There are no known functional issues with the application. However the usability could be improved by enhancing the design and providing a more interactive manual or tooltips. The latest developments can be seen on the project source code repository at https://github.com/urmastalimaa/reactive-visualizer.

The project is currently used at SaleMove Inc. for teaching and learning reactive programming. After improving the usability of Reactive Visualizer it could be proposed to be a resource on the RxJS resources list.

# 6 Related Work

## 6.1 rxvision

rxvision [37] is a visualizer for RxJS developed by Jared Forsyth.

Its goal is to "visualize and debug your RxJS reactive streams"[37]. It is able to simulate real-world applications using actual asynchronous code and the graphs that rxvision produces are highly informative.

However rxvision only supports a small number of operators by overwriting RxJS code with custom proxies. It also needs an actual web page to be integrated into and is not usable on its own. rxvision is also not very helpful for reactive programming beginners as it needs an existing RxJS program to visualize.

In conclusion, rxvision might be of interest to developers who are starting to incorporate RxJS to a web-page, but it does not meet the requirements set for Reactive Visualizer.

## 6.2 RxSpy

RxSpy [38] is a debugger for Rx.NET developed by Markus Olsson.

Its goal is to give the developers a view of all the observables in their application and the signals that they produce. It consists of a small debugger application that must be integrated to the system and an external user interface to navigate and inspect the observables. [38]

RxSpy is also able to deal with actual asynchronous code by capturing the signals produced by Observables. It supports inspecting all observables created through standard Rx operators [38].

Olsson notes that RxSpy struggles with real world application load and is mostly suitable for demos and teaching [38]. Although it fills a different role than Reactive Visualizer, it seems to be very useful for .NET developers learning or teaching Reactive Extensions.

# Conclusion

The goal of this thesis was to reduce the effort of learning reactive programming by the means of creating a beginner-friendly, web-based learning tool called Reactive Visualizer.

A study of asynchronous programming in the web browser listed problems with standard solutions, which reactive programming aims to solve. An investigation of reactive programming using the Reactive Extensions for JavaScript [1] (RxJS) library listed the benefits of RxJS, as well as the difficulties of learning it.

An interactive learning tool, Reactive Visualizer, was designed and implemented by the author of this thesis to tackle some of the problems of learning reactive programming. Reactive Visualizer allows its users to visualize how asynchronous processes can be combined using the operators that RxJS offers. The tool is fully interactive, providing a code builder which allows the user to specify exactly the operators, arguments and objects that he or she is interested in. While doing so Reactive Visualizer stays beginner-friendly by providing sensible defaults and ensuring correct program structure.

Reactive Visualizer was written with robustness and extensibility in mind. Its modules and user interface components are tested and easy to understand. Its components are reusable, allowing Reactive Visualizer to be extended for different languages or reactive programming libraries.

In the end the goal of the thesis was accomplished, as a stable and usable deployment of Reactive Visualizer is available at https://urmastalimaa.github.io/reactive-visualizer/.

## Future work

Though Reactive Visualizer is usable and stable in its current state, its design could be enhanced and its usability improved.

Reactive Visualizer could be presented to students in programming-related courses to introduce them to reactive programming basics. Feedback from the users could also be used to further improve on the current design.

Reactive Visualizer currently only supports visualizing asynchronous behaviour which originates from RxJS operators. One feature that has already been suggested is the ability to capture and replay actual asynchronous behaviour (e.g. web requests).

# A Marble Diagrams of Operators

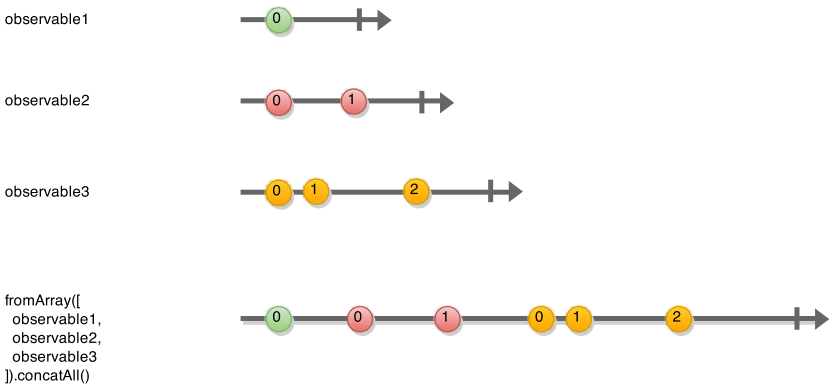


Figure 11: Operator `concatAll` applied to an array three Observables

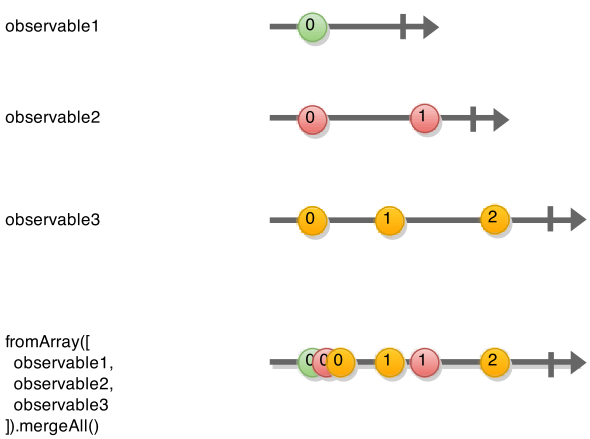


Figure 12: Operator `mergeall` applied to an array three Observables

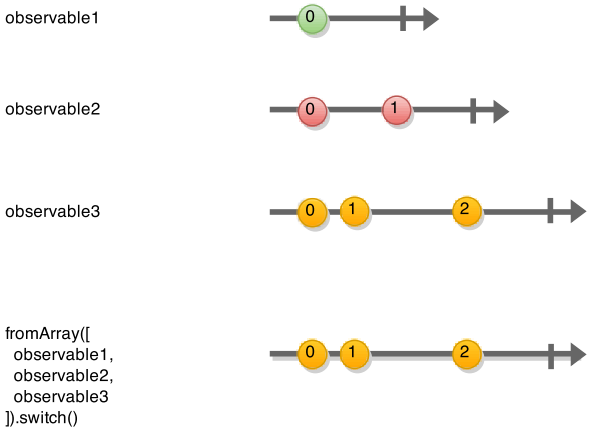


Figure 13: Operator `switch` applied to an array three Observables

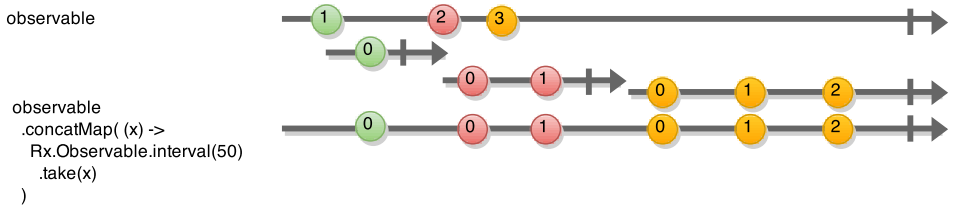


Figure 14: Operator `concatMap` applied to an Observable

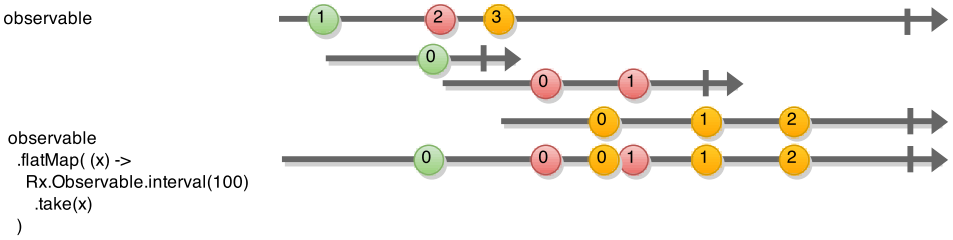


Figure 15: Operator `flatMap` applied to an Observable

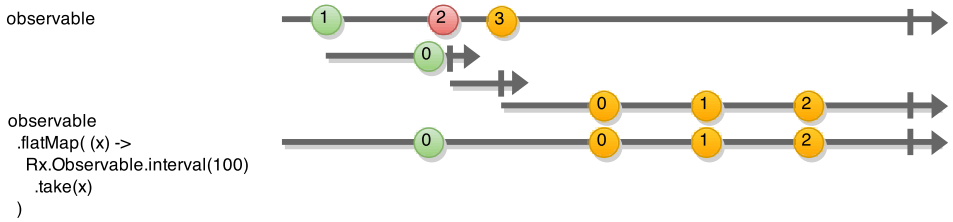


Figure 16: Operator `flatMapLatest` applied to an Observable

# B In-Browser Manual

## Reactive Visualizer

Welcome to using Reactive Visualizer.
Reactive Visualizer was created as a tool to aid in learning reactive programming using the Observable pattern.

### Parts

The tool consists of four distinct parts:

- this manual,
- Observable editor,
- inspector,
- virtual time control,
- persistence.

## Observable Editor

### Purpose

The Observable editor provides a JavaScript editor for creating an Observable. The building blocks for creating an Observable are factories and operators.

### Factory

An Observable is created by specifying a factory, which produces the initial values.
A list of factories is provided in the drop-down next to `Rx.Observable.`.
The simplest Observable consists of a single factory:

```
Rx.Observable.just(5)
```

### Operators

An Observable can be transformed by applying an operator to it. An Observable of 5 values every 500ms is created with:

```
Rx.Observable.interval(500)
  .take(5)
```

A list of operators is provided in the drop-down which opens when hovering over the `+` symbol.
The `+` symbol is located next to existing factories and operators.
An operator can be removed by clicking on the `-` symbol.

### Arguments

The arguments for any building block can be provided by as a JavaScript expression.
There is always a default expression provided.
When making changes, the return type of the expression must remain same.

### Nested Observables

Some building blocks can have arguments which themselves are Observables.
In that case a nested Observable editor is created.

## Inspector

### Purpose

The inspector shows values at a specified time for every Observable building block in the Observable editor.

### Time

Above the Observable editor is row containing integers which denote the virtual time in milliseconds.
Sometimes the integers might seem to be off by one, this is due to simulating the Observable in virtual time.

### Values

The values leaving the Observable building blocks appear next to them.
Faintly colored visual clues help to identify which operators the values originate from.
Note that only values **leaving** an operator are displayed.

### Completion

When some part of the Observable *completes* a green `C` is displayed in the inspector.

### Error

When an error occurs in a part of the Observable the error message text is displayed in the inspector in red.

### Multiple values

When multiple values come from an operator at the exact same time, then they are displayed next to each other.
One unit of time is surrounded by a little border.

### Previous values

As new values appear, older ones are shifted to the left.
The most recent value is always the rightmost.

### Changes to the Observable

Whenever the Observable is changed in the editor, it is analyzed and the values are updated in the inspector immediately.

## Virtual Time Controller

### Purpose

The virtual time controller allows setting the internal clock of the Observable.

### Controlling the Clock

There are two ways to control the internal clock of the Observable.

***Play***
The *Play* button simulates the observable in real-time.

**Slider**
The slider allows setting the internal clock of the Observable simulation to a specific time.

## Persistence

### Purpose

Persistence allow saving and loading example Observables using the *Load* and *Save* buttons.

**Loading** Clicking on the *Load* button opens a modal with the following panels.

**Bundled examples**
Bundled examples are examples which are bundled with the Reactive Visualizer and cannot be changed.

**User examples**
User examples contain examples saved by the user.

**Loading an Example**
Every example is described by its name, which appears on the left side,
and a short description which appears on the right.
To load any example, click on its name.
To delete an example, click on the small red `x` button on the right side of the example.

**Saving an Example**
The *Save* button stores the current Observable in User examples.
When clicked, two mandatory inputs are presented:

- *Name*,
- *Description*.

These values will appear correspondingly in User examples.

# C Initial Reviews of Reactive Visualizer

“ Reactive Visualizer is a simple yet powerful tool for composing and visualizing asynchronous event-based constructions by means of Reactive Extensions. Reactive Visualizer provides the blocks from Rx and allows to combine them in a comprehensive way in order to visualize the results and intermediate processes in a timeline fashion.

As a result one can use Reactive Visualizer to easily grasp the concepts and ideas around reactive programing, debug and troubleshoot complex existing applications' logic, quickly draft/fiddle constructions to later integrate in industry apps, etc.

The tool uses state of the art technologies and is quite robust in terms of Rx capabilities supported and the way that one can define its own JS snippets to be executed along with the data streams. Despite being in early stages and having room for improvement in terms of usability it has proven its value in the development of complex applications that require synchronization and orchestration of different sources of data/information.

As an example, in SaleMove the tool has proven been useful and has become part of the development toolset for developing a complex engaging and communication platform. ” (Carlos Paniagua, CTO of SaleMove Inc.)

“ Reactive Visualizer is a one of a kind tool for learning and testing out functional code. It is extremely helpful for programmers who have no prior experience with functional programming or the observable pattern.

I really like the interactive time track! ” (Siim Halapuu, UX/UI developer at SaleMove Inc.)

“ A very helpful tool for visualizing reactive programming concepts, particularly operators.

Even straightforward looking operators can be deceptively easy to misuse, let alone more complex ones which are nearly impossible to understand from abstract descriptions and documentation examples alone. Especially to programmers, who more often than not, are used to stateful synchronous programming.

Despite lots of existing useful documentation, it's not trivial for beginners to test out and truly understand reactive extensions reactive concepts. Among other things one tends to overlook the significance of schedulers. Just testing the code out real-time is often not enough and might require excessive amount of debugging output to make sense.

This is where this tool shines. I was also pleasantly surprised that input supported objects.

The UI, while minimal, can be inconvenient to use, due to smallness of some elements. Particularly the + and ? signs. Also the placement of text-areas between parenthesis can be hard to follow on more-complex examples, but this isn't an matter to solve in the first place. All in all, a much needed and useful tool, especially for people just starting to try out RxJS. ” (Kait Kasak, Master's student at Tartu University)

# References


[1] Microsoft Open Technologies, Inc. *The Reactive Extensions for JavaScript*. 2015. URL: https://github.com/Reactive-Extensions/RxJS (visited on 05/10/2015).

[2] Facebook, Inc. *React - A JavaScript library for building user interfaces.* 2015. URL: https://facebook.github.io/react/ (visited on 05/10/2015).

[3] Jeremy Ashkenas and open source developers. *CoffeeScript*. 2015. URL: http://coffeescript.org/ (visited on 05/10/2015).

[4] Rodrigo Fonseca. *Introduction to Asynchronous Programming*. 2014. URL: http://cs.brown.edu/courses/cs168/f14/content/docs/async.pdf (visited on 05/10/2015).

[5] Antero Taivalsaari et al. "Web browser as an application platform: The Lively Kernel experience". In: (2008).

[6] Jesse James Garrett et al. "Ajax: A new approach to web applications". In: (2005).

[7] Ian Fette and Alexey Melnikov. "The websocket protocol". In: (2011).

[8] Adam Bergkvist et al. "Webrtc 1.0: Real-time communication between browsers". In: *Working draft, W3C* 91 (2012).

[9] Mozilla Developer Network. *Concurrency model and Event Loop*. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop (visited on 05/10/2015).

[10] Joyent, Inc. and other Node contributors. *Node.js*. 2015. URL: https://nodejs.org/ (visited on 05/10/2015).

[11] Facebook, Inc. *React Native*. 2015. URL: https://facebook.github.io/react-native/ (visited on 05/10/2015).

[12] John Hughes. "Why functional programming matters". In: *The computer journal* 32.2 (1989), pp. 98–107.

[13] Mozilla Developer Network. *Promise object specification*. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (visited on 05/10/2015).

[14] Mozilla Developer Network. *addEventListener specification*. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener (visited on 05/10/2015).

[15] Don Syme, Tomas Petricek, and Dmitry Lomov. "The F# asynchronous programming model". In: *Practical Aspects of Declarative Languages*. Springer, 2011, pp. 175–189.

[16] Wikipedia, the Free Encyclopedia. *Reactive Programming*. 2015. URL: http://en.wikipedia.org/wiki/Reactive_programming (visited on 05/10/2015).

[17] Conal M. Elliott. "Push-pull Functional Reactive Programming". In: *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*. ACM, 2009, pp. 25–36.

[18] Jesse Liberty, Paul Betts, and Stefan Turalski. *Programming Reactive Extensions and LINQ*. Springer, 2011.

[19] Dmitry Soshnikov. *JavaScript Array "Extras" in Detail*. 2011. URL: https://dev.opera.com/articles/javascript-array-extras-in-detail/ (visited on 05/10/2015).

[20] Microsoft Open Technologies, Inc. *RxJS Design Guidelines*. 2015. URL: https://github.com/Reactive-Extensions/RxJS/tree/master/doc/designguidelines (visited on 05/10/2015).

[21] Microsoft Open Technologies, Inc. *Observable object*. 2015. URL: https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md (visited on 05/10/2015).

[22] Microsoft Open Technologies, Inc. *Operators by Categories*. 2015. URL: https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/categories.md (visited on 05/10/2015).

[23] Open source developers. *Operators By Category*. URL: http://reactivex.io/documentation/operators.html (visited on 05/10/2015).

[24] Open source developers. *Observable*. URL: http://reactivex.io/documentation/observable.html (visited on 05/10/2015).

[25] Microsoft Open Technologies, Inc. *Rx.Scheduler class*. 2015. URL: https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/schedulers/scheduler.md (visited on 05/10/2015).

[26] Rolf Molich and Jakob Nielsen. "Improving a Human-computer Dialogue". In: *Commun. ACM* 33.3 (Mar. 1990), pp. 338–348.

[27] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[28] Artem Marchenko, Pekka Abrahamsson, and Tuomas Ihme. "Long-term effects of test-driven development a case study". In: *Agile Processes in Software Engineering and Extreme Programming*. Springer, 2009, pp. 13–22.

[29] Max Schäfer. "Refactoring tools for dynamic languages". In: *Proceedings of the Fifth Workshop on Refactoring Tools*. ACM. 2012, pp. 59–62.

[30] Jeremy Ashkenas and open source developers of the CoffeeScript language. *List of languages that compile to JS*. 2015. URL: https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS (visited on 05/10/2015).

[31] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. "An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. DYLA '13. ACM, 2013, 3:1–3:9.

[32] Facebook, Inc. *Why React?* 2015. URL: https://facebook.github.io/react/docs/why-react.html (visited on 05/10/2015).

[33] Joyent, Inc. and other Node contributors. *Browserify*. URL: http://browserify.org/ (visited on 05/10/2015).

[34] Scott Sauyet, Michael Hurley and open source developers. *Ramda - A practical functional library for Javascript programmers*. 2015. URL: http://ramdajs.com (visited on 05/10/2015).

[35] Microsoft Open Technologies, Inc. *Rx.TestScheduler class*. 2015. URL: https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/testing/testscheduler.md (visited on 05/10/2015).

[36] Mozilla Developer Network. *Web Storage API*. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (visited on 05/10/2015).

[37] Jared Forsyth. *rxvision - Take off the blindfold*. 2015. URL: https://github.com/jaredly/rxvision (visited on 05/10/2015).

[38] Markus Olsson. *RxSpy*. 2014. URL: https://github.com/niik/RxSpy (visited on 05/10/2015).

**Non-exclusive licence to reproduce thesis and make thesis public**

I, Urmas Talimaa (date of birth: 12th of April 1990),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Reactive Visualizer: A Learning Tool for Reactive Programming Using Reactive Extensions for JavaScript

supervised by Aivar Annamaa

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 10.05.2015