

RxJs 1/4

ReactiveX for Javascript

1 EINFÜHRUNG	2
1.1 v6 vs. v5	2
1.2 Datenströme	2
1.3 Programmiersprachen	2
1.4 Vorteile/Nachteile	3
1.5 Marble Diagrams	3
1.5.1 Beispiel CombineLatest	4
2 GRUNDELEMENTE	5
2.1 Observable	5
2.1.1 Wert	5
2.1.2 Error	5
2.1.3 Complete	6
2.2 Observer	6
2.2.1 Beispiel RxJs	6
2.2.2 hot/cold	6
2.2.3 shared/not shared	7
2.3 Subscription	7
2.4 Subject	7
2.5 Vergleich Designpattern Observer	7
3 HELLO WORLD	8
3.1 index.html	8
3.2 index.js	8
3.2.1 Operatoren	8
3.2.2 Präfix	8
3.2.3 Explizit	9
3.2.4 Best practice?	9
4 RXJS VISUALIZER	10
4.1 Überblick	11
4.2 Plain HTML	11
4.2.1 References Plain Javascript	12
4.2.2 init	12
4.2.3 startVisualize	12
4.2.4 Observer	12
4.2.5 pipeable operator draw	12
4.3 Symbols	13
4.3.1 useRandomSymbolsForNumber	13
4.4 Stream creation functions	13

1 Einführung

RxJs ist die Implementierung von ReactiveX für Javascript. Definition auf <http://reactivex.io/>:

„ReactiveX is a combination of the best ideas from the **Observer pattern**, the Iterator pattern, and functional programming.”

Bei Reaktiver Programmierung reagiert man auf Datenströme aller Art, meist arbeitet man aber mit asynchronen Datenströmen. Das Konzept ist dabei ähnlich wie bereits von LINQ bekannt: Man registriert Operatoren und Listener auf diese Datenströme und reagiert entsprechend.

ReactiveX bietet dabei eine Vielzahl an Funktionen, mit denen man derartige Datenströme bearbeiten kann. Die einzelnen Funktionen heißen zwar meist anders als bei LINQ, mit Kenntnissen von LINQ sollte der Einstieg aber kein allzu großes Problem darstellen.

Die Beschreibung für Javascript findet man in <https://rxjs-dev.firebaseapp.com/>

1.1 v6 vs. v5

Dieses Tutorial wurde mit RxJs 6 erstellt. Im Gegensatz zu RxJs 5 wurde dabei das method chaining größtenteils durch die Funktion **pipe()** ersetzt, was aus meiner Sicht den Einstieg erschwert. Außerdem wurden einige wenige Operatoren umbenannt.

Beispiel: RxJs5 → RxJs 6

```
Rx.Observable.fromEvent(updateButton, 'click')
  .map(_ => parseInt(input.value))
  .do(v => (currentNumber = v))
  .startWith(currentNumber)
  .subscribe(observer);
```

```
Rx.Observable.fromEvent(updateButton, 'click').pipe(
  map(_ => parseInt(input.value)),
  tap(v => (currentNumber = v)),
  startWith(currentNumber)
)
.subscribe(observer);
```

1.2 Datenströme

Als Datenstrom kann man sich alles Mögliche vorstellen bzw. diese aus allerlei Konstrukten erzeugen. Somit ist RxJs ziemlich breit anwendbar:

- Variable
- Arrays
- Benutzereingaben
- Events
- Web requests
- ...

1.3 Programmiersprachen

ReactiveX ist in den meisten gängigen Programmiersprachen verfügbar, wobei sich die einzelnen Methoden aber leicht unterscheiden können:

- C#
- Java
- Javascript
- Swift
- Dart
- C++
- Python
- PHP
- ...

In diesem Dokument wird nur RxJs besprochen. Mit diesem Wissen sollte aber auch der Einsatz von ReactiveX in anderen Programmiersprachen schnell einsetzbar sein.

Bei der Dokumentation der einzelnen Funktionen werden unten immer die sprachspezifischen Varianten aufgelistet (<http://reactivex.io/documentation/operators/debounce.html>):

Language-Specific Information:

[RxClojure](#)[RxCpp](#)[RxGroovy `debounce throttleWithTimeout`](#)[RxJava 1.x `debounce throttleWithTimeout`](#)[RxJava 2.x `debounce throttleWithTimeout`](#)[RxJS `debounce debounceWithSelector throttleWithTimeout`](#)[RxKotlin `debounce throttleWithTimeout`](#)[RxNET `Throttle`](#)[RxPHP `throttle`](#)[RxPY `debounce throttle_with_selector throttle_with_timeout`](#)[Rxrb](#)[RxScala `debounce throttleWithTimeout`](#)[RxSwift `debounce throttle`](#)

In den jeweiligen Links sind dann meist auch konkrete Beispiele zu sehen.

1.4 Vorteile/Nachteile

Reactive Programming bietet viele **Vorteile**:

- die Operationen sind nur von Inputs abhängig
- es entstehen keine Seiteneffekte → ist leicht testbar
- es werden darin praktisch keine Variablen angelegt → weniger Fehler

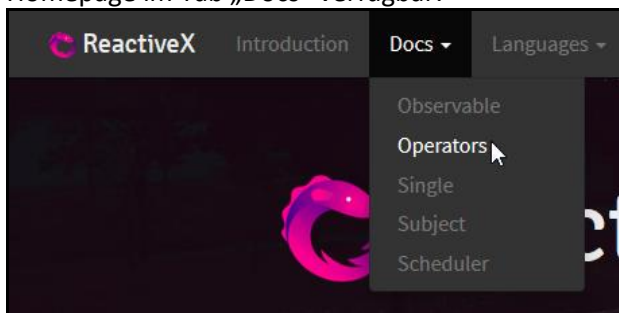
Daraus ergibt sich aber auch ein gravierender **Nachteil**:

- RxJs-Konstrukte sind schwer zu debuggen, weil es ja nur wenige bis keine Variablen gibt

Zum Vergleich: Schleifen berechnen selbst keine Werte, sondern man braucht eben eine Variable, in der z.B. eine Summe berechnet wird.

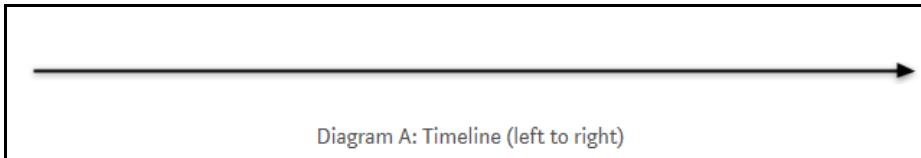
1.5 Marble Diagrams

Die Beschreibung von ReactiveX-Funktionen erfolgt oft graphisch mit „Marble Diagrams“. Diese sind auf der Homepage im Tab „Docs“ verfügbar:

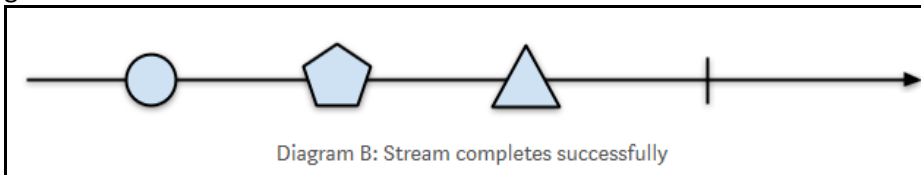


Dabei wird der Ablauf auf einer oder mehreren Timelines beschrieben. Die einzelnen Elemente des Datenstroms werden als Kreis, Rechteck, usw. dargestellt. Siehe z.B. <https://medium.com/@jshvarts/read-marble-diagrams-like-a-pro-3d72934d3ef5>:

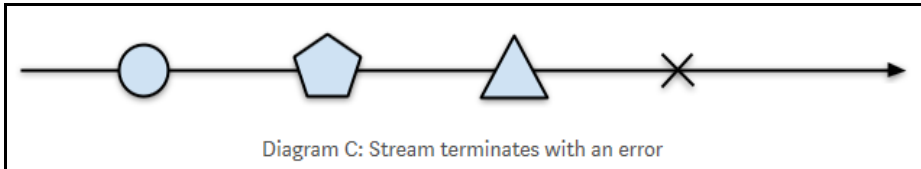
Der Ablauf ist von links nach rechts dargestellt:



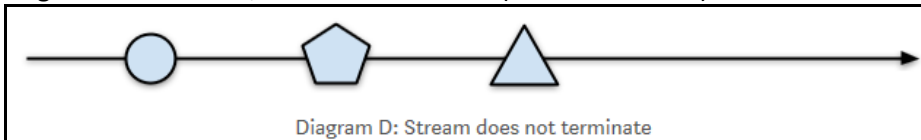
Einzelne Elemente werden auf die Timeline gelegt. Ein Ende des Streams wird mit einem senkrechten Strich gekennzeichnet:



Das Auftreten eines Fehlers wird durch ein X markiert:



Es gibt auch Streams, die nie terminieren (z.B. Klick-Events) – bei diesen fehlt als das „Completed“-Symbol:

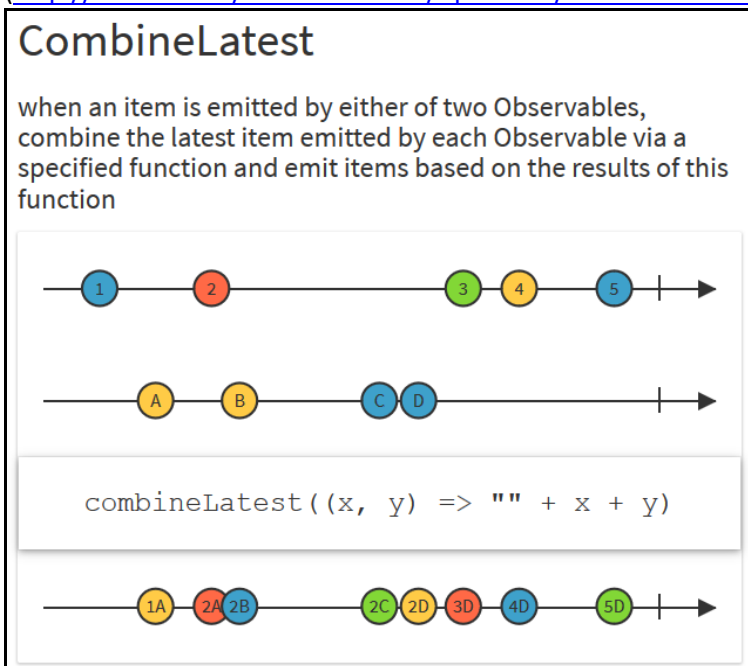


Die Timeline kann man sich vorstellen wie ein Laufband bei LINQ.

1.5.1 Beispiel CombineLatest

Am Beispiel CombineLatest sieht das dann so aus

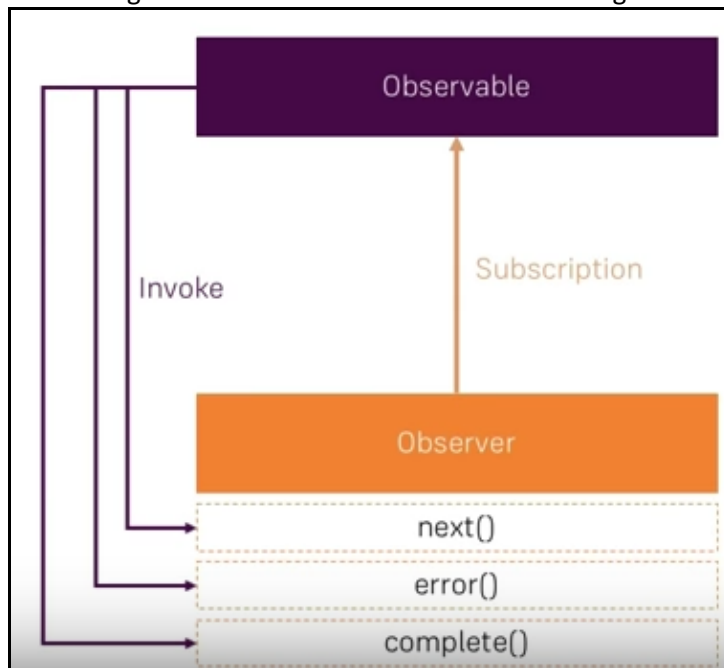
(<http://reactivex.io/documentation/operators/combinelatest.html>):



Die meisten Operatoren werden auch auf der Webseite <http://rxmarbles.com/> dargestellt.

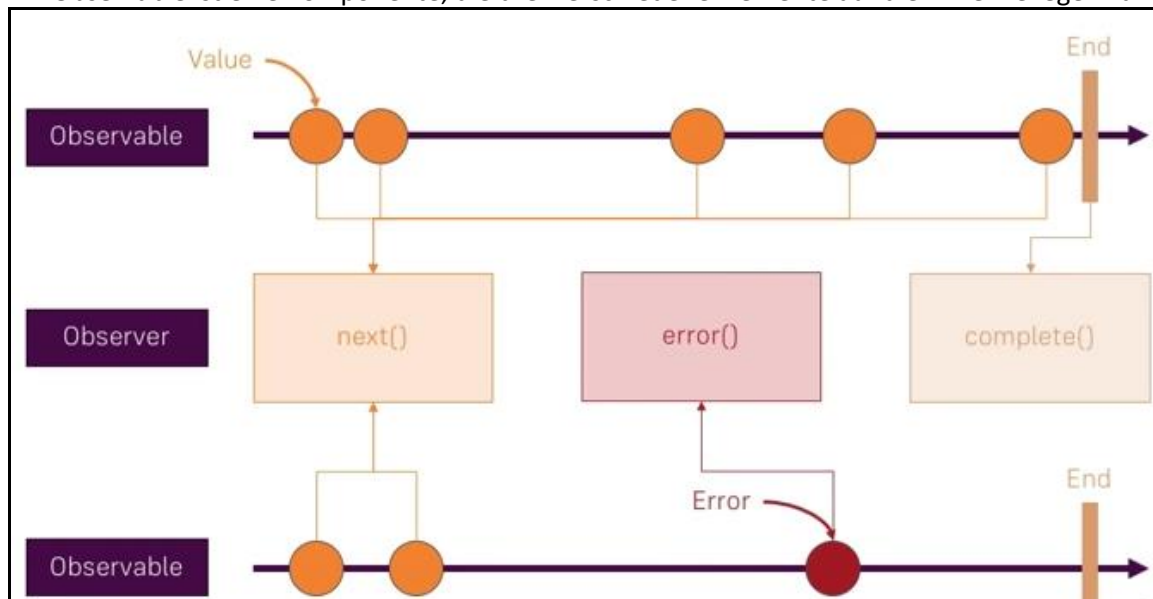
2 Grundelemente

Die wichtigsten Klassen von RxJs sind die in den folgenden Kapiteln beschrieben.



2.1 Observable

Ein Observable ist eine Komponente, die drei verschiedene Elemente auf die Timeline legen kann.



Diese werden jetzt kurz erklärt.

2.1.1 Wert

Ein Wert wird mit **next()** erzeugt. Werte können beliebig oft auf die Timeline gelegt werden.

2.1.2 Error

Fehler werden mit **error()** erzeugt. Es kann maximal ein Error produziert werden, sehr oft wird aber nie ein Error erzeugt – entspricht also am ehesten einer Exception in herkömmlichen Programmen. Nach dem ersten Error bricht die „Produktion“ ab und es werden keine weiteren Werte erzeugt.

2.1.3 Complete

Das explizite Ende der Produktion wird durch **complete()** gemeldet. Danach können weder Werte noch Fehler erzeugt werden. Diese Meldung muss aber nicht erzeugt werden, und sie fehlt auch häufig. In manchen Situationen kann sie gar nicht entstehen, z.B. bei der Produktion von MouseMove-Events, bei denen ja kein Ende erkannt werden kann.

2.2 Observer

Ein Observer ist ein Objekt, das sich bei einem Observable registriert. Da ein Observable entweder ein **Objekt** liefert, einen **Fehler** meldet oder das **Ende** des Streams mitteilt, besteht ein Observer aus untenstehenden drei Methoden, wobei die letzten beiden optional sind.

2.2.1 Beispiel RxJs

Zum besseren Verständnis der erwähnten Methoden wird als Vorgriff ganz kurz die Verwendung in RxJs vorgestellt.

Es wird ein Observable erzeugt (wie das genau geht, sehen wir weiter unten und ist vorerst nicht wesentlich):

```
const observable = ...;
```

2.2.1.1 Mit Callbacks

Darauf kann man sich dann auf Werte registrieren:

```
observable.subscribe(value => console.log(value));
```

Möchte man auch auf Fehler oder Completed reagieren, kann man das mit zwei weiteren Callbacks tun:

```
observable.subscribe(  
  value => console.log(value),  
  error => console.error(error),  
  () => console.log('Completed')  
);
```

2.2.1.2 Mit Observer-Objekt

Anstelle der Callbacks kann man auch ein Observer-Objekt erzeugen und dieses registrieren. Dabei muss der Observer die Properties **next**, **error** und **complete** als Funktionen zur Verfügung stellen:

```
const observer = {  
  next: value => console.log(`next: ${value}`),  
  error: error => console.error(error),  
  complete: () => console.log('Completed')  
};
```

```
observable.subscribe(observer);
```

2.2.2 hot/cold

Ähnlich wie bei LINQ stellt sich die Frage, wann ein Observable Werte zu produzieren beginnt. Das kann man nicht allgemein beantworten, weil es zwei Arten von Observables gibt:

- **hot**: ein derartiges Observable beginnt sofort, Werte zu emittieren, egal ob es Listener gibt, oder nicht. D.h. ein Subscriber kann sich nicht sicher sein, dass er alle Werte des Observables auch tatsächlich bekommt (diese könnten ja schon vor seiner Registrierung emittiert worden sein)
Beispiele:
 - Live-Konzert. Wenn man später kommt, wird der Beginn nicht extra wiederholt und man hat diese Information versäumt.
 - Mouseclicks
- **cold**: produziert erst dann, wenn sich jemand darauf registriert, d.h. wenn mindestens ein **subscribe()** erfolgt ist. Das ist also genau so wie bei LINQ, wo ja auch erst durch **ToList**, **foreach**, **First**,... das damals erwähnte „Laufbandsystem“ eingeschaltet wird.
Beispiele:

- ein Film, der als „Video on demand“ konsumiert wird. Verschiedene Konsumenten können zu beliebigen Zeitpunkten den Film starten
- Webrequest

2.2.3 shared/not shared

Ebenfalls aufpassen muss man, wenn sich mehrere Observer auf ein Observable registrieren, weil nicht automatisch gewährleistet ist, dass sich diese Observer die genau gleichen Werte teilen.

Details siehe weiter unten.

2.3 Subscription

Das Registrieren eines Listeners bei einem Observable nennt man Subscription. Der Aufruf von **subscribe()** liefert ein Objekt vom Typ Subscription, mit dem man sich wieder vom Observable abmelden kann. Das erfolgt durch **unsubscribe()**:

```
let subscription = observable.subscribe(observer);
subscription.unsubscribe();
```

2.4 Subject

Ein Subject implementiert sowohl die Funktionen von **Observable** als auch von **Observer**. Daher kann man ein Subject mit **next()** Werte emittieren, andererseits auch auf Werte horchen.

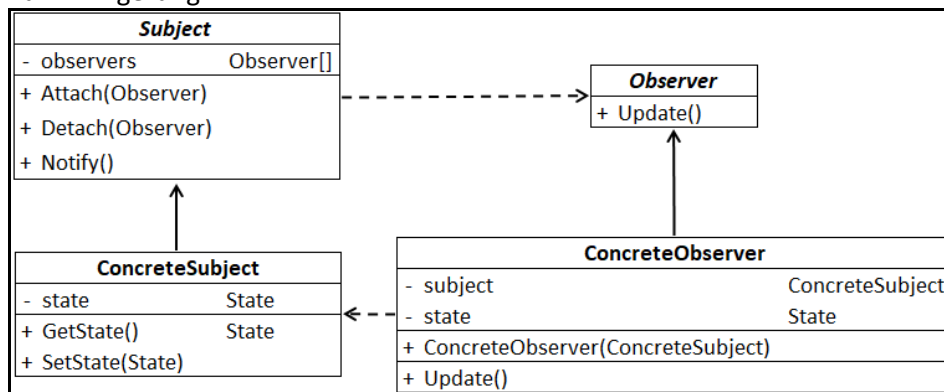
Details dazu werden weiter unten an konkreten Beispielen präsentiert.

2.5 Vergleich Designpattern Observer

RxJs könnte man mit dem Designpatter Observer so vergleichen:

RxJs	Design Pattern
Subject	Observable
Attach()	subscribe()
Detach()	Subscription unsubscribe
Notify()	next() (bzw. error() und complete())
Observer	Observer
Update()	next() (bzw. error() und complete())

Zur Erinnerung:



3 Hello World

Für die Verwendung von RxJs in einer Webseite muss man nur eine einzige Javascript-Datei importieren. Eine CSS-Datei (wie etwa bei Bootstrap) braucht man nicht, weil RxJs ja keine sichtbaren Komponenten enthält sondern nur Datenmanipulationen durchführt.

3.1 index.html

Die Datei **rxjs.umd.js** kann man entweder lokal zur Verfügung stellen oder von einem CDN-System importieren:

```
<html>

<head>
  <meta charset="UTF-8" />
  <title>Hello RxJs</title>
  <script src="lib\jquery\jquery.js"></script>
  <!-- <script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script> -->
  <script src="lib\rxjs-6\rxjs.umd.min.js"></script>
  <script src="tutorial.js"></script>
</head>

<body>
  <h1>Check Console for output...</h1>
  <div id="logs">Output comes here</div>
</body>

</html>
```

In der lokalen Variante ist es prinzipiell egal, wo die Datei liegt, es bietet sich aber an, eine gewisse Struktur einzuhalten, bei der man die Version am Pfad erkennen kann:

	Name	Änderungsdatum	Typ	Größe
Tutorial				
lib				
jquery				
rxjs-6				
rxjs-visualizer				
	_readme.txt	20.02.2019 18:03	TXT-Datei	1 KB
	rxjs.umd.js	15.05.2019 11:25	JS-Datei	368 KB
	rxjs.umd.min.js	20.02.2019 18:01	JS-Datei	123 KB
	rxjs.umd.min.js.map	08.05.2019 09:34	Linker Address Map	284 KB

Wie bei praktisch allen Frameworks gibt es eine lesbare und eine minified/uglified Variante, die sich vor allem in der Größe unterscheiden. Einmal die min-Datei öffnen, dann ist klar was mit minified und uglified gemeint ist.

3.2 index.js

In index.js soll jetzt RxJs verwendet werden. Dabei sind die Objekte über das Präfix **Rx** aufrufbar. Die Operatoren sind dann auf ein Observable direkt verwendbar.

In index.js soll jetzt RxJs verwendet werden. RxJs ist modular aufgebaut (darauf weist das Infix **umd** im Dateinamen hin), entsprechend müssen die einzelnen Objekte und Funktionen importiert werden.

Dabei hat man zwei Möglichkeiten (das hat aber nichts mit RxJs zu tun sondern ist bei Javascript aufgrund von Destructuring grundsätzlich so).

3.2.1 Operatoren

Die Operatoren müssen in beiden Fällen von **rxjs.operators** importiert werden:

```
const { map, filter } = rxjs.operators;
```

3.2.2 Präfix

```
const Rx = rxjs;
```


Dadurch kann man alle Funktionen mit **Rx. xxx** verwenden. Man importiert also alles aus einem Modul und stellt es über ein Präfix zur Verfügung.

```
Rx.range(1, 10)
  .pipe(
    filter(x => x % 2 === 1),
    map(x => x + x)
  )
  .subscribe(x => console.log(x));
```

3.2.3 Explizit

Man kann auch einzelne Objekte/Funktionen explizit importieren. Dabei bedient man sich des sogenannten Destructuring.

```
const { Observable, Subject, ReplaySubject, from, of, range, fromEvent } = rxjs;
```

Dann kann man sich das Rx-Präfix sparen, obiges Beispiel mit range würde dann also so aussehen:

```
range(1, 10)
  .pipe(
    filter(x => x % 2 === 1),
    map(x => x + x)
  )
  .subscribe(x => console.log(x));
```

3.2.4 Best practice?

Zur besseren Lesbarkeit wäre eine Möglichkeit, die Klassen explizit zu importieren, und die Funktionen über das Präfix zu verwenden.

```
const Rx = rxjs;
const { Observable, Subject, ReplaySubject } = rxjs;
const { map, filter } = rxjs.operators;
```

4 RxJs Visualizer

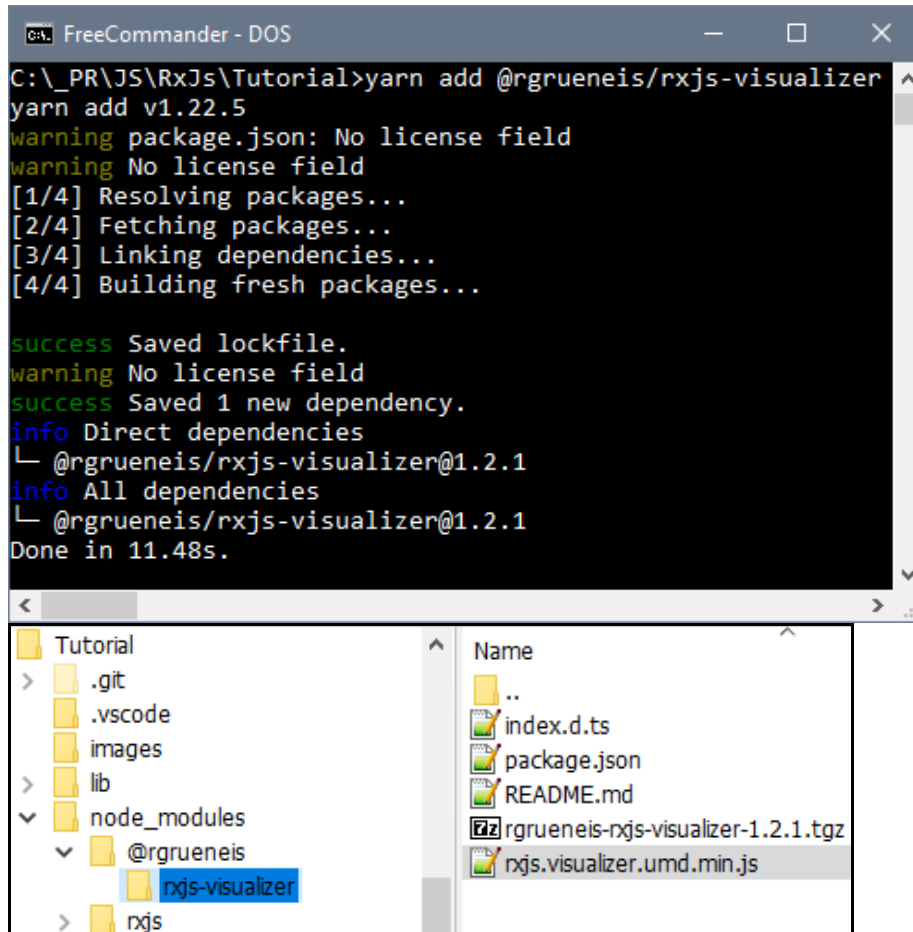
Aufgrund der Asynchronität von RxJs hängt vieles vom zeitlichen Ablauf ab. Damit man sich diese Abläufe grafisch anzeigen lassen kann, habe ich ein Node-Modul programmiert, das eben genau das ermöglicht. Es ist auf <https://www.npmjs.com/package/@rgrueneis/rxjs-visualizer> verfügbar und man kann es mit

```
npm i @rgrueneis/rxjs-visualizer
```

bzw.

```
yarn add @rgrueneis/rxjs-visualizer
```

installieren.



```
C:\_PR\JS\RxJs\Tutorial>yarn add @rgrueneis/rxjs-visualizer
yarn add v1.22.5
warning package.json: No license field
warning No license field
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
warning No license field
success Saved 1 new dependency.
info Direct dependencies
└─ @rgrueneis/rxjs-visualizer@1.2.1
info All dependencies
└─ @rgrueneis/rxjs-visualizer@1.2.1
Done in 11.48s.
```

The file explorer shows the following structure:

- Tutorial
 - .git
 - .vscode
 - images
 - lib
 - node_modules
 - @rgrueneis
 - rxjs-visualizer
 - rxjs

The file explorer also shows the contents of the rxjs-visualizer directory:

- ..
- index.d.ts
- package.json
- README.md
- rgrueneis-rxjs-visualizer-1.2.1.tgz
- rxjs.visualizer.umd.min.js

Dabei wird auch RxJs automatisch als Abhängigkeit mit installiert.

Es ist als UMD-Modul implementiert und kann daher in konventionellen Javascript-Programmen als auch in Angular eingesetzt werden.

Die entsprechende Javascript-Datei muss als Referenz hinzugefügt werden:

```
<head>
  <meta charset="UTF-8" />
  <title>Hello RxJs</title>
  <script src="lib/jquery/jquery.js"></script>
  <script src="node_modules/rxjs/bundles/rxjs.umd.min.js"></script>
  <script src="node_modules/@rgrueneis/rxjs-visualizer/rxjs.visualizer.umd.min.js"></script>
  <script src="tutorial.js"></script>
  <link rel="stylesheet"
        type="text/css"
        href="tutorial.css">
</head>
```

Für die Anzeige benötigt man ein <canvas> und ein <div>:

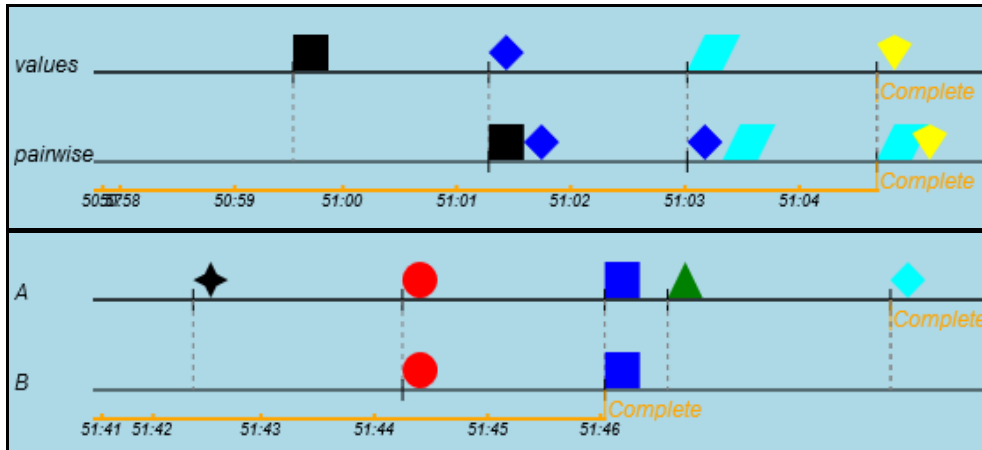
```
<canvas id="canvas"
| | | | |
| | | | |
| | | | |
width="700"
height="300"></canvas>
<div id="logs">Output comes here</div>
```

Auf der NPM-Homepage ist eine genauere Beschreibung, die wichtigsten Teile sind hier kurz angeführt.

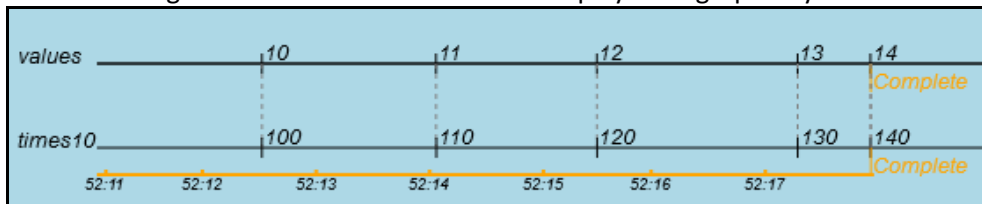
4.1 Überblick

This plugin is intended to illustrate streaming with RxJs.

For this purpose, the resulting values/objects are displayed on a timeline by making special observer objects available.



You can configure whether the elements are displayed as graphic symbols or as texts/values.



4.2 Plain HTML

To use RxVis the following is required:

- RxJs 6.x
- rxjs.visualizer.min.js
- a <canvas>-Element
- optionally a <div> for displaying the logs

So a minimal HTML file could look like this:

```
<html>

<head>
  <meta charset="UTF-8" />
  <title>Hello RxJs</title>
  <script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>
  <script src="rxjs.visualizer.min.js"></script>

  <script src="tutorial.js"></script>
  <link rel="stylesheet" type="text/css" href="tutorial.css">
</head>

<body>
  <canvas id="canvas" width="700" height="300"></canvas>
  <div id="logs">Output comes here</div>
</body>

</html>
```

4.2.1 References Plain Javascript

The following references have to be added to a Javascript file to be able to use RxJs visualization:

```
const { DrawingSymbol } = RxJsVisualizer;
const { draw } = RxJsVisualizer.operators;
```

4.2.2 init

Before using RxJsVisualizer the RxVis-object has to be initialized. This is done with an options object, which in the minimal version looks like this.

```
RxJsVisualizer.init({
  canvasId: 'canvas',
  logDivId: 'logs'
});
```

The ids correspond to the ids of a <canvas> and a <div>, respectively, given in the HTML file.

4.2.3 startVisualize

To start the visual animation the function startVisualize() has to be called. This will clear the canvas and (re)start the timer.

```
RxJsVisualizer.prepareCanvas(['values']);
RxJsVisualizer.startVisualize();
RxJsVisualizer.createStreamFromArraySequence([10, 11, 12, 13, 14])
  .subscribe(RxJsVisualizer.observerForLine(0, 'value', true));
```

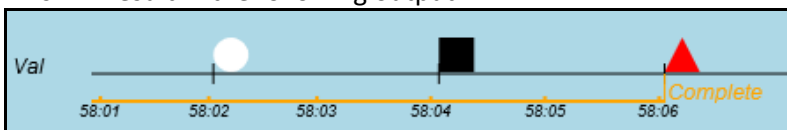
4.2.4 Observer

Two functions are required:

- prepareCanvas: the array given as parameter specifies the number and headings of the various timelines
- observerForLine: specifies the line index, where the objects of the stream are written/drawn.

```
RxJsVisualizer.prepareCanvas(['Val']);
RxJsVisualizer.startVisualize();
Rx.timer(1000, 2000)
  .pipe(take(3))
  .subscribe(RxJsVisualizer.observerForLine(0));
```

This will result in the following output:



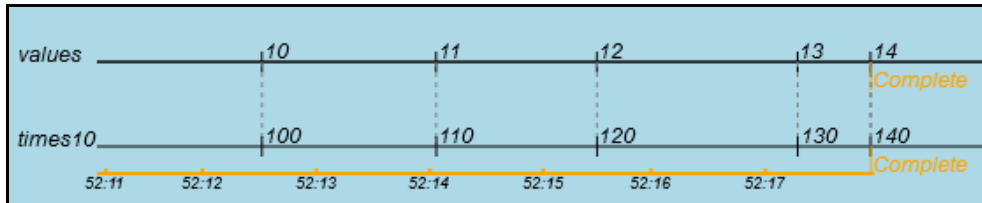
4.2.5 pipeable operator draw

Another possibility to draw to the timeline is using the pipeable operator draw.

Example:

```
RxJsVisualizer.prepareCanvas(['values', 'times10']);
RxJsVisualizer.startVisualize();
RxJsVisualizer.createStreamFromArraySequence([10, 11, 12, 13, 14])
  .pipe(
    draw(0, '', true),
    map(x => x * 10)
  )
  .subscribe(RxJsVisualizer.observerForLine(1, '*10:', true));
```

Result:



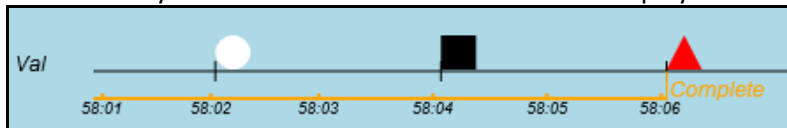
4.3 Symbols

Instead of writing text to the timeline it is often more vivid to display symbols instead. To do so, for any text a symbol can be registered.

```
const symbols = {};
symbols['[object MouseEvent]'] =
    new DrawingSymbol({ color: 'blue', shape: 'circle' });
symbols['0'] = new DrawingSymbol({color: 'white', shape:'circle'});
symbols['1'] = new DrawingSymbol({color: 'black', shape:'square'});
symbols['2'] = new DrawingSymbol({color: 'red', shape:'triangle'});

RxJsVisualizer.init({
  canvasId: 'canvas',
  logDivId: 'logs',
  symbolMap: symbols
});
RxJsVisualizer.useRandomSymbolsForNumbers(100);
```

With these symbols the above RxJs-stream will be displayed like this:



4.3.1 useRandomSymbolsForNumber

As marble diagrams are used quite often, a helper function is available to generate standard symbols for numbers.

The following code generates symbols for all numbers 0-99:

```
RxJsVisualizer.useRandomSymbolsForNumbers(100);
```

4.4 Stream creation functions

Various functions are available, to randomly create random symbols at random intervals. See the below manual for these functions.