R x JS

Prološčić, Nikola

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:123980

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science







Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Nikola Prološčić

RxJS

Završni rad

Osijek, 2019.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Nikola Prološčić

RxJS

Završni rad

Mentor: izv. prof. dr. sc. Domagoj Matijević
Komentor: mag. math. Jurica Maltar

Osijek, 2019.

Sažetak

Tema ovog završnog rada je RxJS biblioteka. Najprije ćemo reći što je Reaktivno programiranje i tok podataka, a potom uvesti pojam klase Observable koja nam omogućuje vršenje raznih operacija nad tokom podataka. Objasnit ćemo dizajn obrasce Observer i Iterator na kojima se Observable temelji. Kreirat ćemo Observable, a potom i sučelje Observer koje implementira 3 metode: `next, complete` i `error`. Nakon toga pretplatit ćemo Observer na Observable i reći nešto više o Supscription-u i njegovoj metodi `unsubscribe`. Uz jednostavne primjere praćene marble-diagramom navest ćemo nekoliko operatora kreiranja Observable-a i Pipable operatora. Pipable operatore možemo vezati `pipe` funkcijom, a pokazat ćemo i kako. Subject je specijalna vrsta Observable-a koji istovremeno može biti tvorac podataka kao Observable i potrošač podataka kao Observer. Kod Subject-a svi pretplatnici dijele isto izvršavanje, što nije slučaj kod standardnog Observable-a, no multicast metodom to možemo promijeniti. Na kraju ćemo objasniti kako pomoću Scheduler-a kontrolirati tijek izvršavanja Observable-a.

Ključne riječi: RxJS, Reaktivno programiranje, Observable, Observer, pretplata, operatori, Subject, Scheduler

Abstract

Subject of this bachelor's thesis is RxJS. First we will tell what is Reactive programming and data stream. After that we will introduce term of class Observable which enables us performing many operations on data stream. We will explain design patterns Observer and Iterator on which Observable is based on. We will create Observable and interface Observer which implements three methods: `next, complete` and `error`. After that we will subscribe Observer on Observable and say something about Subscription and his method `unsubsribe`. With simple examples followed by marble-diagrams we will list few create and Pipable operators. Pipable operators are pipe able, in other words, we can connect them with pipe function and we will show how. Subject is special type of Observable which can be both data producer like Observable and data consumer like Observer. All subscribers of Subject share same execution which is not case with Observable. But with multicast method we can change that. On the end we will explain how to control execution of Observable with Scheduler.

Key words: RxJS, Reactive programming, Observable, Observer, subscribe, operators, Subject, Scheduler

Sadržaj

1 Uvod **1**

2 Reaktivno programiranje **1**

3 Observable **2**
3.1 Observer obrazac . . . 3
3.2 Iterator obrazac . . . 3
3.3 Kreiranje Observable-a konstruktorom . . . 4

4 Observer **4**

5 Subscribe **5**
5.1 Pretplata na Observable . . . 5
5.2 Subscription . . . 5

6 Operatori **7**
6.1 Operatori kreiranja . . . 7
6.1.1 Kreiranje Observable-a iz polja . . . 7
6.1.2 Kreiranje Observable-a iz JavaScript događaja . . . 8
6.1.3 Ostale metode za kreiranje Observable-a . . . 8
6.2 Pipable operatori . . . 9
6.2.1 Operator delay . . . 10
6.2.2 Operator repeat . . . 10
6.2.3 Pipe funkcija . . . 11

7 Subject **11**
7.1 Multicast operator . . . 13

8 Scheduler **14**

9 Zaključak **16**

1 Uvod

RxJS je implementacija Reactive Extensions (skraćeno ReactiveX ili Rx) biblioteke za JavaScript[1]. ReactiveX je biblioteka za izgradnju asinkronih programa zasnovanih na događajima koristeći niz Observable-a (poglavlje 3). Kreirana je od strane Microsofta kako bi omogućila reaktivno programiranje u .NET okviru. ReactiveX je ubrzo prerastao u standardardnu biblioteku svih poznatijih programskih jezika pa tako na službenoj stranici (vidi [3]) uz RxJS možemo naći implementacije za Javu (RxJava), Python (RxPy), C# (Rx.NET) i druge. Konzistentnost je glavna osobina ove biblioteke, princip je za sve implementacije gotovo isti, razlika je samo u sintaksi.
Namjera ovog rada je čitatelju uz jednostavne primjere[2] objasniti osnovne koncepte RxJS biblioteke koji rješavaju upravljanje asinkronim događajima.
A osnovni koncepti su:

- Observable
- Observer
- Subscription
- Operatori
- Subject
- Schedulers

Njih ćemo opisati u poglavljima od 3 do 8. Prvo ćemo reći nešto više o samom reaktivnom programiranju.

2 Reaktivno programiranje

Danas svaka ozbiljnija aplikacija sadrži mnogo pokretnih asinkronih dijelova koje uobičajenim načinom programiranja nije jednostavno koordinirati. Uz to želimo preduhitriti svaki potencijalni problem kao što su: gubitak internetske veze, rušenje servera, greške u samom softveru itd. Tu nastupa reaktivno programiranje.
Reaktivno programiranje (engl. reactive programming) je programska paradigma[3] strukturirana oko tokova podataka. Za razliku od tipičnog načina izrade softvera gdje izričito pišemo kod kako bismo upravljali promjenama nad varijablama spremljenim u memoriji (imperativno programiranje), ovdje gradimo softversku aplikaciju koja će reagirati na promjene koje se događaju. U fokus umjesto varijabli dolazi tok podataka. Tok je niz podataka poredanih u vremenu.Takav koncept najbolje je objasnio tvorac RxJS-a, Erik Meijer riječima: "Tvoj miš je baza podataka" - kako je ilustrirano na slici 1. Na uzastopne klikove miša možemo gledati kao na niz podataka razdvojenih vremenom, a na takav niz elemenata kao tok podataka. Kombiniranjem tokova podataka ponovno dobivamo tok podataka.

[1]JavaScript je jednostavan, interpretiran programski jezik namijenjen ponajprije razvoju interaktivnih HTML-stranica. Može se izvoditi u svim modernim web preglednicima.

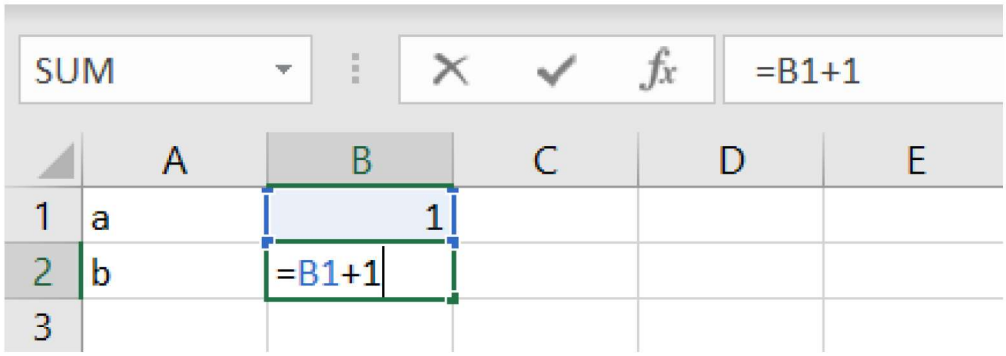
[2]Svi kod primjeri svodit će se na kratki ispis u konzoli.Za kompleksnije primjere pogledati na [6]

[3]Programska paradigma je programski obrazac, stil, način programiranja. Osnovne programske paradigme su: Imperativna, objektno-orijentirana, funkcionalna i logička. Ostale paradigme tretiramo kao podparadigme ili kombinacije osnovnih(Više na [5]).



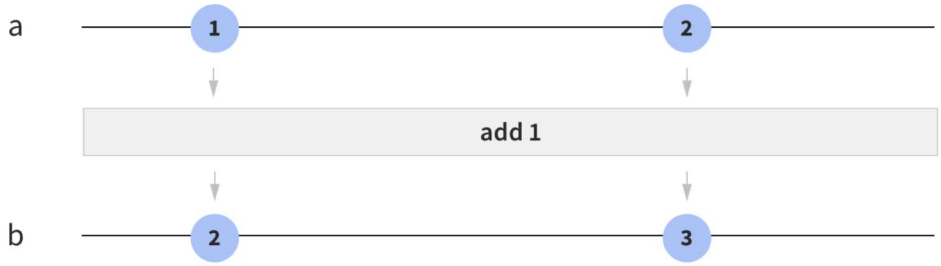
Slika 1: Klik miša kao tok podataka

Primjer reaktivnog programiranja s kojim smo se gotovo svi susreli su osnovne funkcije proračunskih tablica (npr. SUM u Microsoft Excel-u). Vidi sliku 2.



Slika 2: SUM u Excel tablici

Ako vrijednosti ćelije B1 dodamo 1, vrijednost ćelije B2 će se automatski povećati za 1. Ako o a i b razmišljamo kao tokovima podataka onda prethodnu situaciju možemo prikazati kao na slici 3.



Slika 3: Tokovi podataka

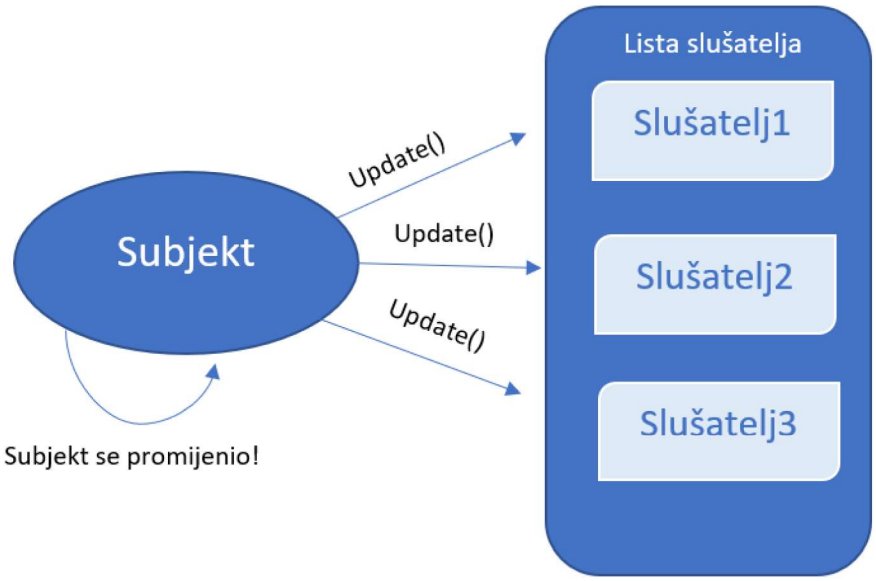
3 Observable

Sada znamo što je tok podataka, no kako iskoristiti taj pojam? Htjeli bismo da nad njim možemo vršiti operacije kao nad poljem - filtrirati ga, pretraživati, dohvatiti svaki element, stvarati nove tokove, itd. Tu dolazimo do pojma *Observable*. Pomoću RxJS-ove klase *Observable* stvaramo nove tokove podataka, obrađujemo ih i emitiramo njegovim pretplatnicima.Tako na objekt klase *Observable* možemo gledati kao na *tok podataka koji može emitirati svoje vrijednosti*. Pretplatnika na Observable nazivamo Observer.

Da bismo lakše razumjeli kako Observable funkcionira, prvo ćemo objasniti dva dizajn obrasca na kojima se temelji.

3.1 Observer obrazac

Observer obrazac se sastoji od glavnog objekta, tvorca ili subjekta i niza slušatelja. *Subjekt* sadrži listu slušatelja pretplaćenih na njega. Pri svakoj promjeni na njemu prolazi kroz listu slušatelja,redom poziva njihove `update()` metode i prosljeđuje im novu vrijednost.



Slika 4: Observer obrazac

3.2 Iterator obrazac

Drugi je Iterator obrazac. On nam omogućuje jednostavan prolazak kroz sadržaj nekog podatka. Potrebne su mu samo dvije metode: `next()` - dohvaćanje sljedeće vrijednosti iz niza i `hasNext()` - provjerava je li preostalo još elemenata u nizu.

Kombinaciju *Observer* i *Iterator* obrasca nazivamo *Rx obrazac*. Observable emitira svoje vrijednosti redom kao iterator, no umjesto da njegovi potrošači zahtjevaju sljedeću vrijednost, *Observable* "gura" vrijednosti potrošačima čim one postanu dostupne. Takva komunikacija *tvorca podataka* (eng.data producer) s *potrošačem podataka* (data consumer) pripada *Push-protokolu*[4].
Za razliku od Subjekta iz Observer obrasca, Observable ne počinje emitirati dok nema barem jednog pretplatitelja, tj. Observer-a. Uz to, kao i iterator, Observable signalizira kada je niz gotov.

[4]U Push-protokolu tvorac određuje kada poslati podatke klijentu, dok klijentu preostaje samo čekati.

3.3 Kreiranje Observable-a konstruktorom

Kreirati Observable možemo na više načina. Za početak ćemo koristiti konstruktor klase Observable kako je prikazano u Kodu 1.

```
1 const observable = new Rx.Observable(function subscribe(observer){
2   observer.next('prvi signal');
3   observer.next('drugi signal');
4   observer.complete();
5 });
```

Kod 1: Kreiranje Observable-a pomoću konstruktora

Konstruktor uzima callback[5] funkciju koja prima *Observer* kao parametar. Funkcijom definiramo kako će *Observable* emitirati vrijednosti promatračima. U ovom primjeru će,nakon što se pretplate na njega, prvo svakom pojedinom Observer-u poslati `string` 'prvi signal', potom `string` 'drugi signal' i onda im javiti da je tok podataka gotov. Radi lakšeg razumijevanja ovdje pišemo kompletan kod, u nastavku ćemo koristiti skraćeni zapis, tzv. "fat arrow" funkciju kao u kodu 2. Analogno kreiramo Observable i sa `create` metodom.
Dobra je praksa pri kreiranju Observable-a umotati sav kod subscribe funkcije unutar `try/catch` bloka koji će odaslati obavijest o grešci ako uhvati ikakvu iznimku (vido kod 2).

```
1 const observable = Rx.Observable.create(observer=>{
2     try{
3         observer.next('prvi signal');
4         observer.next('drugi signal');
5         observer.complete();
6     }catch (err){
7         observer.error(err);
8     }
9   });
```

Kod 2: Kreiranje Observable-a create metodom koristeći `try/catch` blok

4 Observer

Da bismo se pretplatili na Observable koristimo Observer-e.
Observer je generičko sučelje(eng.Interface) koje sadrži 3 metode:

1. next - Ekvivalent update funkciji iz Observer patterna.Pozvana je svaki put kad Observable emitira novu vrijednost.

2. complete - Signalizira da više nema dostupnih podataka. Ne šalje nikakvu vrijednost.

3. error - Pozvan je kada dođe do greške u Observable-u. Šalje JavaScript grešku ili iznimku.

`next` može biti isporučen 0 ili više puta, a `complate` i `error` samo jednom i nakon njih više nema isporuke. Pri inicijalizaciji Observera obavezno je definirati next funkciju, dok su complete i error opcionalne.

[5]Callback je funkcija koja se prosljeđuje drugoj funkciji kao parametar i bit će pokrenuta kada ju glavna funkcija izvrši.

Sučelje Observer kreiramo kao u kodu 3.

```
1  var observer = {
2    next: x => console.log(x),
3    error: err => console.error('Doslo je do greske: ' + err),
4    complete: () => console.log('gotovo'),
5  };
```

Kod 3: Kreiranje Observera

5 Subscribe

5.1 Pretplata na Observable

Sada kada znamo što je Observable i Observer možemo pretplatiti jedan Observer na Observeble pomoću funkcije subscribe kao u kodu 4.

```
1 const subscription= observable.subscribe(observer);
```

Kod 4: Pretplaćivanje na Observable



Slika 5: Tijek Observable-a

Nije slučajno što observable.subscribe i subscribe u konstruktoru `Observable(function subscribe(observer) ...)` imaju isto ime. One jesu različite u biblioteci RxJS, no iz praktičnih razloga možemo ih smatrati jednakim. Kada pozivamo `observable.subscribe` sa proslijeđenim Observer-om, funkcija `subscribe` iz `Rx.Observable(function subscribe(observer)...` je pokrenuta za taj Observer. To nam jamči da `subscribe()` poziv nije dijeljen među različitim Observer-ima istog Observable-a.
Prilikom poziva `observable.subscribe(observer)`, Observer je pridružen listi pretplatnika tog observable-a. No taj poziv vraća i jedan objekt kojeg zovemo *Subscription.*

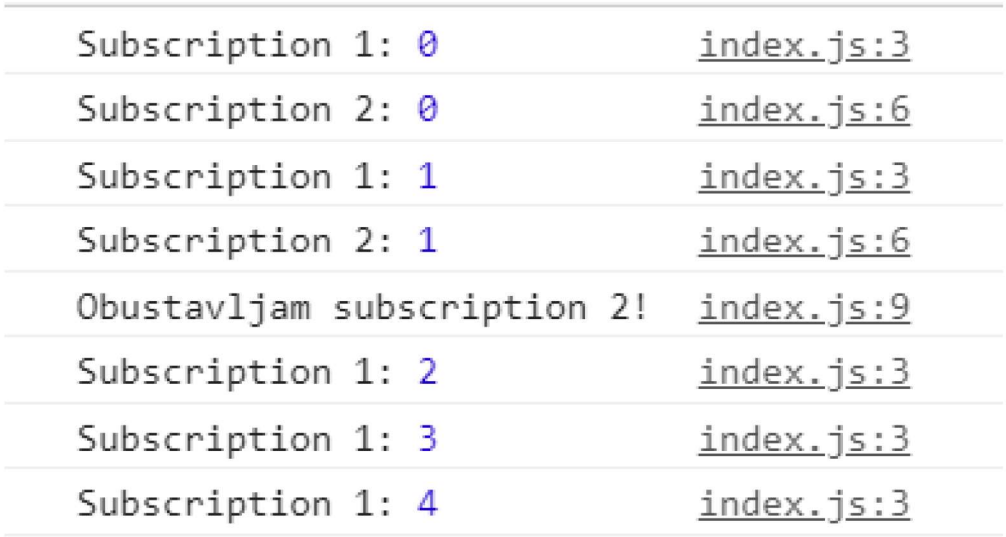
5.2 Subscription

Ponekad izvršavanje Observable-a može biti beskonačno pa je jasno da želimo imati opciju obustave istog, a to nam omogućuje Subscription. *Subscription* je objekt koji reprenzetira izvršavanje Observable-a. Ima samo jednu metodu *unsubscribe.* Budući da je izvršavanje Observable-a za pojedini *Observer* neovisno o drugima, kada pozovemo `subscription.unsubscribe` obustavlja se isporuka sa Observable-a onom Observer-u od kojeg je taj Subscription nastao (vidi kod 5 i sliku 6).

Za potrebe sljedećeg primjera Observable ćemo kreirati pomoću `interval` metode. Intervalom stvaramo Observable koji emitira inkrementirajuće brojeve svakih n milisekundi, gdje je n broj koji smo zadali pri inicijalizaciji. U sljedećem primjeru $n = 1000$.

```
1  var counter = Rx.Observable.interval(1000);
2  var subscription1 = counter.subscribe(function(i) {
3  console.log('Subscription 1:', i);
4  });
5  var subscription2 = counter.subscribe(function(i) {
6  console.log('Subscription 2:', i);
7  });
8  setTimeout(function() {
9  console.log('Obustavljam subscription 2!');
10 subscription2.unsubscribe()
11 }, 2000);
```

Kod 5: Metoda unsubscribe

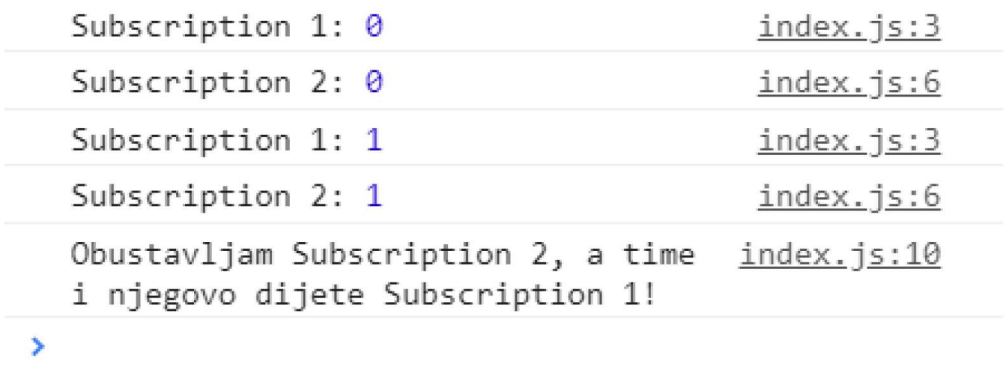


Slika 6: Izvršavanje koda 5

Dogodilo se upravo ono što smo očekivali, no postoji način da ih oba zaustavimo jednom naredbom. Subscription možemo ulančavati. Time si olakšavamo posao jer obustavom roditelja i dijete Subscription će se također obustaviti. Dodavanje jednog subscription-a u drugi činimo naredbom: `Subscription.add(childSubscription)`. kao u kodu 6.

```
1  var counter = Rx.Observable.interval(1000);
2  var subscription1 = counter.subscribe(function(i) {
3  console.log('Subscription 1:', i);});
4  var subscription2 = counter.subscribe(function(i) {
5  console.log('Subscription 2:', i);
6  });
7  subscription2.add(subscription1);
8  setTimeout(function() {
9  console.log('Obustavljam Subscription 2, a time i njegovo dijete
       Subscription 1!');
10 subscription2.unsubscribe()
11 }, 2000);
```

Kod 6: Metoda unsubscribe



Slika 7: Izvršavanje koda 6

6 Operatori

Kao što smo rekli na početku Poglavlja 3, htjeli bismo vršiti operacije nad tokom podataka jednostavno kao nad poljem, a pomoću operatora RxJS biblioteke upravo to i možemo. Oni su najkorisniji dio cijele biblioteke. *Operatori* su funkcije. U RxJS biblioteci postoje dvije vrste operatora.Operatori kreiranja i Pipeable operatori. U ovom poglavlju umjesto primjera izvršenog koda uz kod prilagat ćemo marble diagrame.[6]

6.1 Operatori kreiranja

U 3.3 spomenuli smo da Observable možemo kreirati na više načina i pokazali kako ga kreirati konstruktorom Observable te metodom `Observable.create()`. Puno češće Observable stvaramo iz nekog drugog JavaScript objekta, spajanjem više Observable-a ili operatorima kao što su `interval` (vidi 5.2), `range, timer`...Najčešće ih stvaramo iz polja i događaja.

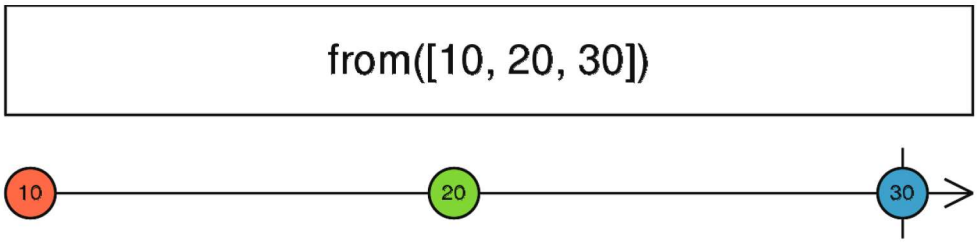
6.1.1 Kreiranje Observable-a iz polja

Možemo kreirati Observable iz bilo kojeg objekta nalik polju, iteratoru, Promisu ili Observable-u koristeći univerzalni `from` operator. **from** uzima polje kao parametar i vraća Observable koji emitira svaki element toga polja (Kod 7.).

```
1 const array = [10, 20, 30];
2 const myObservable = Rx.Observable.from(array);
3
4 myObservable.subscribe(x => console.log(x));
```

Kod 7: Kreiranje Observable-a iz polja brojeva

[6]Marble-diagrami su vizualne reprenzetacije djelovanja operatora. Kod marble-diagrama vrijeme teće prema desno, kuglice (eng.marble) predstavljaju vrijednosti, X grešku tj. `error`, a vertikalna linija kraj izvršavanja tj. `complete`.



Slika 8: Marble diagram koda 7

6.1.2 Kreiranje Observable-a iz JavaScript događaja

Prilikom objašnjavanja pojma tok podataka u Poglavlju 2 dali smo kao primjer uzastopne klikove miša. Od događaja kao što je klik miša Observable kreiramo pomoću operatora `fromEvent`. Operator `fromEvent` prima dva parametra: prvi je element, a drugi ime događaja. Kao element možemo proslijediti neki `DOM EventTarget` (element iz dokumenta npr. Document, Div iz dokumenta, button), JQuerry event target itd., a kao ime događaja: "click", "mousemove", "keypress", "animationstart",..[7].

```
1 const clicks = Rx.Observable.fromEvent(document, 'click');
2 clicks.subscribe(c =>
3     {console.log('x=' + c.clientX + ' y=' + c.clientY)});
```

Kod 8: Kreiranje Observable-a od klikova mišom



Slika 9: Pri svakom kliku u konzoli se ispisuju koordinate gdje je kliknuto

6.1.3 Ostale metode za kreiranje Observable-a

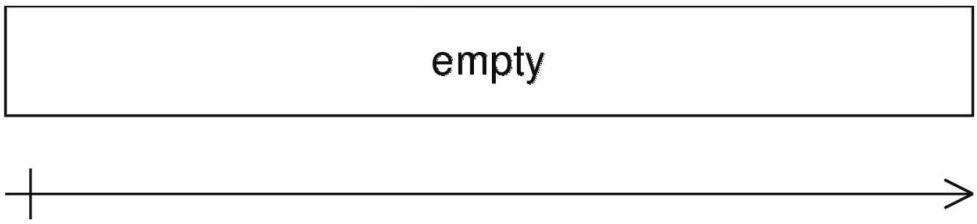
Uz `from`, `fromEvent` i `interval` u operatore kreiranja spadaju:`ajax`, `bindCallback`, `bindNodeCallback`, `defer`, `empty`, `fromEventPattern`, `generate`, `of`, `range`, `throwError`, `timer`, `iif`.
Nešto više reći ćemo još o `empty` i `of`. Za ostale pogledati dokumentaciju na službenoj stranici [4].

[7] popis svih na str. https://developer.mozilla.org/en-US/docs/Web/Events

Operator `empty` ne šalje ni jedan element prema Observeru i odmah pokreće `complete`.

```
1 const result = Rx.Observable.empty()
2 result.subscribe({
3     next: x => console.log(x),
4     complete: () => console.log('gotovo')});
```

Kod 9: Kreiranje Observable-a empty metodom

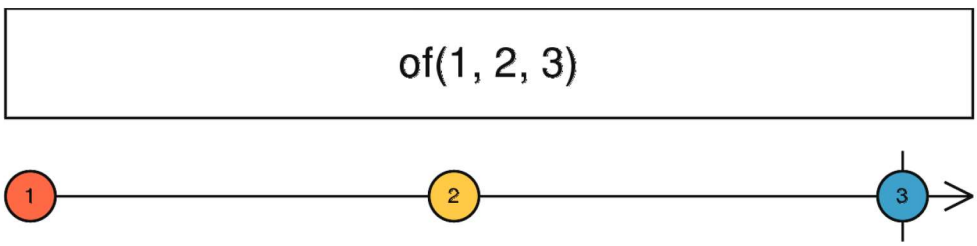


Slika 10: Observable nastao empty metodom

Iako nalikuje na `from` operator `of` ne prima polje, nego svaku prosljeđenu vrijednost gleda u cijelini. Ako mu proslijedimo samo jedno polje, emitirat će samo jednu vrijednost, upravo to polje. Ako želimo dobiti isti Observable kao iz primjera za from, umjesto polja proslijedit ćemo elemente tog polja:

```
1
2 const myObservable = Rx.Observable.of(10, 20, 30);
3
4 myObservable.subscribe(x => console.log(x));
```

Kod 10: Kreiranje Observable-a metodom of



Slika 11: Marble diagram koda 9

6.2 Pipable operatori

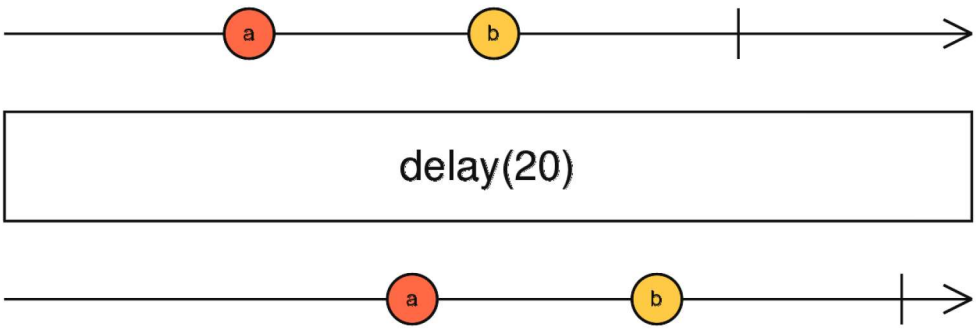
Pipable operator je čista funkcija koja prima Observable kao parametar, a vraća novi Observable. Kažemo da je čista jer pri njenom izvršavanju početni Observable ostaje nepromijenjen. Mogućnosti koje nam pružaju Pipable operatori su nepregledne. U nastavku ćemo obraditi operatore **delay** i **repeat**. Popis ostalih kao i njihovu dokumentaciju potražiti na službenoj stranici [4]. Dodatne primjere korištenja istih potražiti na [6].

6.2.1 Operator delay

Operator `dalay` vraća novi Observable u kojem je emitiranje podataka iz početnog Observable-a odgađeno za n milisekundi gdje je n proslijeđeni parametar (ili do prosljeđenog datuma).

```
1 import {delay } from 'rxjs/operators';
2 import { of } from 'rxjs';
3
4 const delayedThing = delay(20)(of('a','b'));
5 delayedThing.subscribe((v) => console.log(v));
```

Kod 11: Primjer djelovanja delay operatora



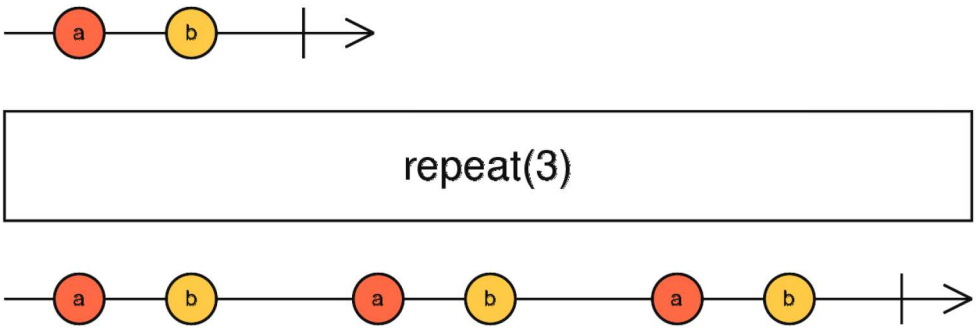
Slika 12: Operator delay iz koda 11

6.2.2 Operator repeat

Operator `repeat` operator vraća novi Observable gdje su sve emitirane vrijednosti početnog Observable-a ponavljene n puta, pri ćemu je n prosljeđeni parametar.

```
1 import {repeat } from 'rxjs/operators';
2 import { of } from 'rxjs';
3
4 const repeatedThing = repeat(3)(of('a','b'));
5 repeatedThing.subscribe((v) => console.log(v));
```

Kod 12: Primjer djelovanja repeat operatora



Slika 13: Marble-diagram repeat operatora iz koda 12

6.2.3 Pipe funkcija

Na prethodna dva primjera vidjeli smo kako od postojećeg Observable-a načiniti novi. No što ako bismo htjeli povezati ta dva primjera tako da na početni Observable primijenimo prvo delay pa repeat, ali bez da rezultat delay operatora spremamo u memoriju? Imali bismo ovako nešto:

```
1 const delayAndRepeat = repeat(3)(delay(20)(of('a','b')));
```

Ovakav zapis je teško čitljiv, a još je teže uočiti grešku ako se pojavi. No RxJS ima riješenje za ovaj problem, a zove se *pipe funkcija*. Pomoću nje možemo vezati pipable operatore,odakle slijedi i samo ime vrste[8]. Pomoću pipe funkcije gornji problem riješavamo kao u kodu 13.

```
1 import { repeat, delay } from 'rxjs/operators';
2 import { of } from 'rxjs';
3
4 const delAndRepeat = of('delayed value').pipe(delay(2000),repeat(3));
5 delAndRepeat.subscribe(console.log);
```

Kod 13: Korištenje pipe funkcije

Pipe funkciju možemo možemo koristiti i kad imamo samo jedan operator. Tako umjesto

```
1  const  delayedThing = delay (20)(of('a','b'));
```

pišemo:

```
1  const delayedThing = of('a','b').pipe(delay(2000));
```

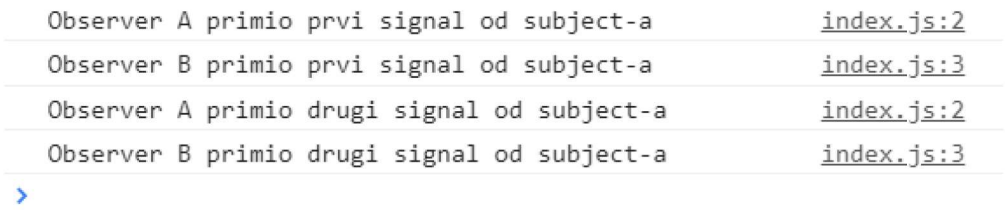
7 Subject

Subject je specijalna vrsta Observable-a koji istovremeno može biti tvorac podataka kao Observable i potrošač podataka kao Observer. *Subject* može emitirati podatke kao *Observable* i istovremeno kao *Observer* biti pretplaćen na neki drugi Observable (vidi kod 14).

```
1 var subject = new Rx.Subject();
2 subject.subscribe(v =>console.log('ObserverA primio '+v+' od subject-a'));
3 subject.subscribe(v =>console.log('ObserverB primio '+v+' od subject-a'));
4
5 var observable = Rx.Observable.of('prvi signal', 'drugi signal');
6
7 observable.subscribe(subject);
```

Kod 14: Subject kao tvorac i potrošač podataka

[8] eng.able-moguće

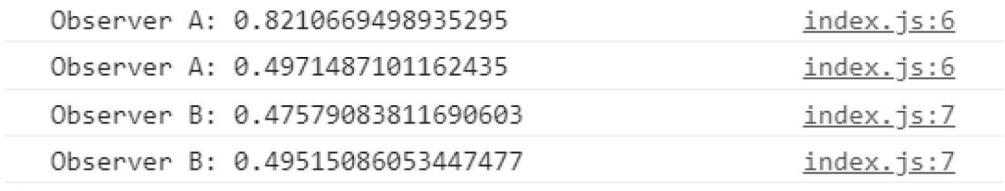


Slika 14: Izvršavanje koda 14

U kodu 14 možemo primijetiti da su oba Observera pretplaćena na Subject primila istu vrijednost. Iz toga možemo zaključiti da svi pretplatnici Subject-a dijele isto izvršavanje (vidi kod 16 i sliku 16), što nije slučaj kod pretplate na Observable (vidi kod 15 i sliku 15). Možemo reći da je Subject specijalna klasa Observable-a koja omogućuje podacima višesmjerno odašiljanje (eng.multicasting[9]).

```
1  var observable = Rx.Observable.create(observer => {
2      observer.next(Math.random()),
3      observer.next(Math.random());
4      });
5
6    observable.subscribe(v => console.log('Observer A: ' + v));
7    observable.subscribe(v => console.log('Observer B: ' + v));
```

Kod 15: Funkcija math.random odaslana direktno preko Observable-a

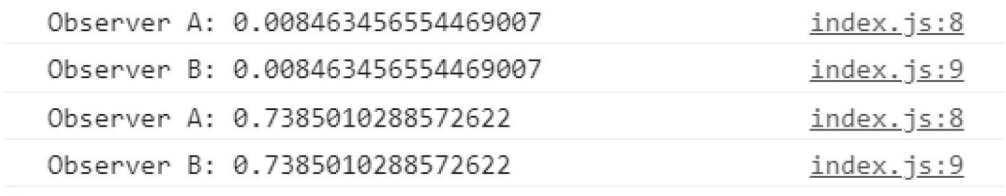


Slika 15: Za svaki pretplaćeni Observer izvršio se zaseban Observable

```
1  var observable = Rx.Observable.create(observer => {
2      observer.next(Math.random()),
3      observer.next(Math.random());
4    });
5
6    var subject = new Rx.Subject();
7
8    subject.subscribe(v => console.log('Observer A: ' + v));
9    subject.subscribe(v => console.log('Observer B: ' + v));
10
11   observable.subscribe(subject);
```

Kod 16: Funkcija math.random odaslana preko Subject-a

[9]Za standardni Observable kažemo da je jednosmjeran (eng.unicasted).

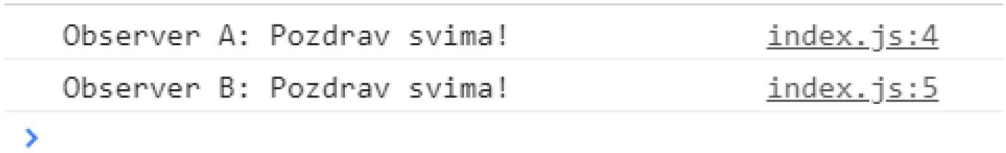


Slika 16: Pretplatnici Subject-a dijele isto izvršavanje

Pošto je Observer, Subject također sadrži metode `next, error i complate`. Zbog toga u svakom trenutku možemo metodom `next(value)` odaslati svim pretplatnicima novu vrijednost kao što to činimo u kodu 17.

```
1
2 var subject = new Rx.Subject();
3
4 subject.subscribe(v => console.log('Observer A: ' + v));
5 subject.subscribe(v => console.log('Observer B: ' + v));
6
7 subject.next('Pozdrav svima!');
```

Kod 17: Poziv Subject.next metode



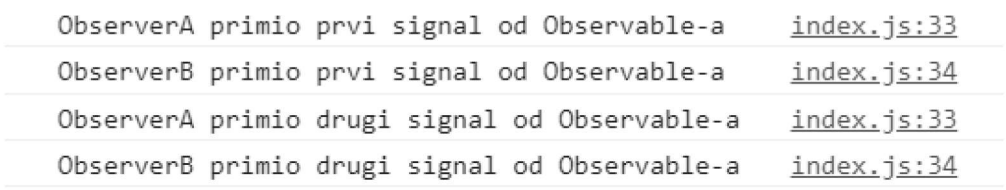
Slika 17: Izvršavnje koda 17

7.1 Multicast operator

Observable klasa ima `multicast` metodu koja jednosmjerni Observable pretvara u višesmjerni. Pozivamo ju nad Observable-om, a prosljeđujemo joj Subject. Tako se možemo pretplatiti direktno na Observable kao u kodu 18 i dobiti isti rezultat kao pretplatom na Subject u kodu 14. Metodom `multicast` Observable pretvaramo u `ConnectableObservable` čije izvršavanje ne počinje sve dok ne pozovemo `connect()` metodu.

```
1 var observable = Rx.Observable
2         .of('prvi signal', 'drugi signal')
3         .multicast(new Rx.Subject);
4
5 observable.subscribe(v =>console.log('ObserverA primio '+ v +
6                                       ' od Observable-a'));
7 observable.subscribe(v =>console.log('ObserverB primio '+ v +
                                       ' od Observable-a'));
8
9 observable.connect();
```

Kod 18: Metoda `multicast(subject)`



Slika 18: Izvršavnje koda 18

Kod Subject-a i višesmjernih Observable-a jednom kad isti završe s emitiranjem, tj. nakon poziva `complete()` metode, naknadno pretplaćeni Observer neće primiti podatke. Takav slučaj imamo s Observerom `C` u kodu 19.

```
1  var observable = Rx.Observable.interval(500).take(4);
2
3  var sub = new Rx.Subject();
4  var source = observable.multicast(sub);
5
6  source.subscribe(x => console.log('A : ' + x));
7
8  source.connect();
9
10 setTimeout(() => {
11   source.subscribe(x => console.log('B : ' + x));
12 }, 1200);
13
14 setTimeout(() => {
15   source.subscribe(x => console.log('C : ' + x));
16 }, 2400);
```

Kod 19: Zakašnjeli pretplatnik



Slika 19: A je primio sve, B dio, a C je zakasnio

8 Scheduler

Možemo primjetiti da se izvršavanje Observable-a neće isto odvijati kada je Observable kreiran metodom `from` i metodom `interval`. Kada se Observer pretplati na Observable kreiran `from` metodom, Observable će mu odmah isporučiti sve podatke i završiti, a program će nastaviti dalje. No kada se Observer pretplati na Observable nastao `interval` metodom isti će mu podatke isporučivati asinkrono, usporedno s izvršavanjem glavnog programa.

Na službenoj stranici [4] među detaljnim informacijama o operatoru `from` navedeno je kako `from` kao parametre obavezno prima objekt sličan Observable-u (Observable, array, Promise,..), a kao opcionalni parametar Scheduler tipa SchedulerLike kojemu je zadana vrijednost *undefined*. Ako isto potražimo za operator `interval` vidjet ćemo da je u njegovom slučaju za `Scheduler` zadana vrijednost *async*.
Scheduler kontrolira kada će izvršavanje, tj. Subscription započeti i na koji način će Observable odašiljati podatke svojim Observer-ima. Tako pomoću `Async` Scheduler-a postavljamo izvršavanje Observable-a kao asinkrono. U kodu 20 i na slici 20 možemo vidjeti što će se dogoditi ako `from` metodi proslijedimo `asyncScheduler` kao drugi parametar.

```
1  import { from, asyncScheduler } from 'rxjs';
2
3  console.log('start');
4
5  const array = [10, 20, 30];
6  const result = from(array, asyncScheduler);
7
8  result.subscribe(x => console.log(x));
9
10 console.log('end');
```

Kod 20: Proslijeđivanje asyncScheduler parametra `from` metodi

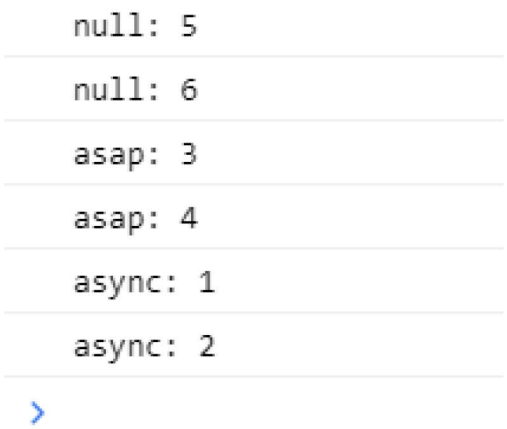
```
start
end
10
20
30
>
```

Slika 20: Asinkrono izvršavanje Observable-a nastalog `from` metodom

Izborom različitih Scheduler-a kontroliramo brzinu i redoslijed emitiranja podataka s Observable-a prema njegovim Observer-ima. Sa funkcijom SubscribeOn izabiremo koju vrstu Scheduler-a će Observable koristiti (vidi kod 21).

```
1  import { of, asapScheduler, asyncScheduler} from 'rxjs';
2  import { subscribeOn } from 'rxjs/operators';
3
4  const a = of(1, 2).pipe(subscribeOn(asyncScheduler));
5  const b = of(3, 4).pipe(subscribeOn(asapScheduler));
6  const c = of(5, 6);
7
8  a.subscribe(v=>console.log('async: '+ v));
9  b.subscribe(v=>console.log('asap: '+v));
10 c.subscribe(v=>console.log('null: '+ v));
```

Kod 21: Korištenje `subscribeOn` funkcije



Slika 21: Koristeći `asapScheduler` dobivamo asinkroni način, ali brži od `asyncScheduler`

Uz `null, asyncScheduler i asapScheduler` RxJS nam nudi `queueScheduler` te `animationFrameScheduler`.

9 Zaključak

Reaktivno programiranje je postalo sastavni dio mnogih grana programiranja, posebice razvoja web stranica i aplikacija. Razvojom interneta kvaliteta informacija koje pružamo na svojim web stranicama više nije dovoljna, važno je i kako ih prezentiramo. Korisnik neće biti zadovoljan ako mu informacije ponudimo na način kako je to bilo prije 10 godina. Moramo pratiti trendove i tražiti nova rješenja. Ponekad rješenja koja nudi sam programski jezik u kojem vršimo razvoj, npr. Javascript, nisu dovoljna, a i ako jesu znaju biti dosta komplicirana. Zbog njezine jednostavnosti i prisutnosti u svim većim programskim jezicima, poznavanje RxJS biblioteke je nužno kako bi bili konkurentni na tržištu.

Literatura

[1] Sergi Mansilla, *Reactive Programming with RxJS*, The Pragmatic Programmers, 2015.

[2] Paul P. Daniels, Luis Atencio *RxJS in Action, 1st Edition* Manning Publications, 2017.

[3] ReactiveX, *Supported languages*, `http://reactivex.io/languages.html`

[4] RxJS, *Reactive Extensions Library for JavaScript*, `https://rxjs-dev.firebaseapp.com/`

[5] University of Central Florida, Department of CS, *Major Programming Paradigms* `http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html`

[6] LernRxJS, *RxJS Operators By Example* `https://www.learnrxjs.io/operators/`