

<https://www.tutorialspoint.com/rxjs/index.htm>

RxJS - Overview

Az RxJS teljes neve: Reactive Extension for Javascript. Ez egy javascript könyvtár, amely Observable elemeket használ az reaktív programozással, amely aszinkron adathívásokkal, callback-el és eseményalapú programokkal (event-based programs) foglalkozik. Az RxJS más Javascript könyvtárakkal és keretrendszerekkel együtt használható. Javascript és TypeScript-et is támogatja.

What is RxJS?

Az RxJS hivatalos honlapjának megfelelően könyvtárként definiáljuk aszinkron és eseményalapú programok összeállítását observable szekvenciák felhasználásával. Az egyik alapvető típust, a Observable, satellite types (Observer, Schedulers, Subjects) és az Array#extras (map, filter, reduce, every, etc.) ihlette operátorokat. Biztosítja, hogy az aszinkron események gyűjteményként (collection) kezelhetők legyenek.

Features of RxJS

Az RxJS-ben a következő fogalmak gondoskodnak az aszinkron feladat kezeléséről -

Observable

Observable egy olyan funkció, amely observer-t hoz létre és csatolja a forráshoz, ahol az értékek várhatók, például kattintások, egér események egy dom elemtől vagy egy Http kérés stb.

Observer

Ez egy objektum a next(), error() és complete() metódusokkal, amelyet akkor hívnak meg, ha kölcsönhatásba lép az observable-el, azaz a forrás kölcsönhatásba lép egy példa kattintásra, Http kérésre stb.

Subscription

Amikor az observable létrejön, az observable végrehajtásához fel kell iratkoznunk rá. Használható a végrehajtás törlésére is.

Operators

Az operator pure function, amely az observable-t bemenetként veszi fel, és a kimenet is observable.

Subject

A subject observable, amely képes multicast küldésre, azaz sok observable-el beszélgetni. Vegyünk egy gombot egy event listener-rel, az eseményhez csatolt függvényt az addlistener segítségével hívjuk meg minden alkalommal, amikor a felhasználó rákattint a gombra, hasonló funkciók vonatkoznak az alanyra is.

Schedulers

A scheduler ellenőrzi a subscription kezdetének és értesítésének végrehajtását.

When to use RxJS?

Ha a projekt sok aszinkron feladatkezelésből áll, akkor az RxJS jó választás. Alapértelmezés szerint az Angular projekttel van betöltve.

Advantages of using RxJS

- Az RxJS más Javascript könyvtárakkal és keretrendszerekkel együtt használható. Javascript és Typescript is támogatja. Néhány példa: Angular, ReactJS, Vuejs, nodejs stb.
- Az RxJS egy fantasztikus könyvtár az aszinkron feladatok kezelésében. Az RxJS observables elemeket használ az aszinkron adathívásokkal, callback és eseményalapú programokkal foglalkozó reaktív programozással.
- Az RxJS operátorok hatalmas gyűjteményét kínálja matematikai, transzformációs, szűrési, utility, feltételes, hibakezelési, join kategóriákban, ami megkönnyíti az életet, ha reaktív programozással használják.

Disadvantages of using RxJS

- A kód hibakeresése observable-el nehéz.
- Amint elkezdi az Observable használatát, a teljes kódot a observables alá csomagolhatja.

RxJS - Environment Setup

- NodeJS
- Npm
- RxJS package installation

```
npm install ---save-dev rxjs
```

RxJS - Observables

Observable egy olyan függvény, amely observer-t hoz létre és csatolja a forráshoz, ahol értékeket várnak el, például kattintások, egér események egy dom elemtől vagy egy Http kérés stb.

Az Observer egy olyan callback függvénnyel rendelkező objektum, amelyet akkor hívnak meg, ha interakció van az Observable-el, azaz a forrás kölcsönhatásba lépett egy kattintással, Http kéréssel stb.

Create Observable

Az observable létrehozható observable konstruktorral, observable create metódussal és a subscribe függvény argumentumként történő átadásával az alábbiak szerint:

testrx.js

```
import { Observable } from 'rxjs';

var observable = new Observable(
  function subscribe(subscriber) {
    subscriber.next("My First Observable")
  }
);
```

Létrehoztunk egy Observable-t és hozzáadtunk egy "My First Observable" üzenetet az subscriber.next metódussal, amely az Observable-on belül elérhető.

Az Observable-t az Observable.create () módszerrel is létrehozhatjuk az alábbiak szerint –

testrx.js

```
import { Observable } from 'rxjs';

var observer = Observable.create(
  function subscribe(subscriber) {
    subscriber.next("My First Observable")
  }
);
```

Subscribe Observable

testrx.js

```
import { Observable } from 'rxjs';

var observer = new Observable(
  function subscribe(subscriber) {
    subscriber.next("My First Observable")
  }
);

observer.subscribe(x => console.log(x));
```

Amikor az Observer feliratkozott, megkezdte a Observable végrehajtását.

Execute Observable

Egy observable végrehajtásra kerül akkor, ha feliratkozunk rá. Az observer egy olyan objektum, amelynek három módszere van értesítve,

- `next()` - Ez a módszer olyan értékeket küld, mint egy szám, karakterlánc, objektum stb.
- `teljes()` - Ez a módszer nem küld értéket, és az observable-t befejezettnek jelzi.
- `error()` - Ez a módszer elküldi a hibát, ha van ilyen.

testrx.js

```
import { Observable } from 'rxjs';
var observer = new Observable(
  function subscribe(subscriber) {
    try {
      subscriber.next("My First Observable");
      subscriber.next("Testing Observable");
      subscriber.complete();
    } catch(e) {
      subscriber.error(e);
    }
  }
);
observer.subscribe(x => console.log(x), (e)=>console.log(e),
  ()=>console.log("Observable is complete"));
```

RxJS - Operators

Az operátorok az RxJS fontos részét képezik. Az operator pure függvény, amely observable bemenetként veszi fel, és a kimenet is observable.

Working with Operators

Az operátorokkal való együttműködéshez szükségünk van egy `pipe()` metódusra.

```
let obs = of(1,2,3); // an observable
obs.pipe(
  operator1(),
  operator2(),
  operator3(),
  operator3(),
)
```

A fenti példában létrehoztunk egy observable-t `of()` metódussal, amely felveszi az 1., 2. és 3. értékeket. Most ezen az observable-n különböző műveleteket hajthat végre tetszőleges számú operátor felhasználásával a `pipe()` módszerrel, a fentiek szerint. Az operátorok végrehajtása a observable értékek alapján folytatódik.

```
import { of } from 'rxjs';
```

```
import { map, reduce, filter } from 'rxjs/operators';

let test1 = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
let case1 = test1.pipe(
  filter(x => x % 2 === 0),
  reduce((acc, one) => acc + one, 0)
);
case1.subscribe(x => console.log(x));
```

Kimenet: 30

A fenti példában olyan filter operátort használtunk, amely kiszűri a páros számokat, majd pedig a reduce () operátort használtuk, amely hozzáadja a páros értékeket és megadja az eredményt subscribe-kor.

Az alábbi observable operátorok (típus szerint) vannak:

- Creation
- Mathematical
- Join
- Transformation
- Filtering
- Utility
- Conditional
- Multicasting
- Error handling

Creation Operators

Sr.No	Operator & Description
1	ajax ajax kérést küld a megadott URL-re.
2	from Ez az operátor egy tömbből, egy tömbszerű objektumból, egy promise-ből, egy iterálható objektumból vagy egy observable-szerű objektumból hoz létre observable-t.
3	fromEvent Ez az operátor observable-ként adja meg a kimenetet, amelyet az eseményeket kibocsátó elemeken kell használni, például gombok, kattintások stb.
4	fromEventPattern Ez az operátor observable-t hoz létre az eseménykezelők regisztrálásához használt bemeneti függvényből.
5	interval Ez az operátor minden alkalommal létrehoz egy observable-t a megadott időre.
6	of

	Ez az operátor átveszi az átadott argumentumokat, és observable-é alakítja azokat.
7	range Ez az operátor létrehoz egy observable-t, amely a megadott tartomány alapján számsorozatot ad.
8	throwError Ez az operátor létrehoz egy observable-t, amely értesíti a hibát.
9	timer Ez az operátor létrehoz egy observable értéket, amely az időtúllépés után kiadja az értéket, és az érték folyamatosan növekszik minden hívás után.
10	iif Ez az operátor dönti el, hogy melyik observable fogja előfizetni (subscribe).

RxJS - Creation Operator Ajax

```
import { map, retry } from 'rxjs/operators';
import { ajax } from 'rxjs/ajax';

let final_val = ajax('https://jsonplaceholder.typicode.com/users').pipe(
  map(e => e.response)
);
final_val.subscribe(x => console.log(x));
```

RxJS - Creation Operator from

```
from(input: ObservableInput): Observable
```

- input - Az operátornak adott bemenet observable.
- Visszatérési érték -Observable eredményt ad.

```
import { from } from 'rxjs';

let arr = [2, 4, 6, 8, 10];
let test = from(arr);
test.subscribe(x => console.log(x));
```

RxJS - Creation Operator fromEvent

```
fromEvent(target: eventtarget, eventName: string): Observable
```

- target - A cél a dom elem
- eventName - eventName, amelyet rögzíteni szeretne, például kattintás, egérmutató stb.

```
import { fromEvent, interval } from 'rxjs';
import { buffer } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let interval_events = interval(1000);
let buffered_array = interval_events.pipe(buffer(btn_clicks));
buffered_array.subscribe(arr => console.log(arr));
```

RxJS - Creation Operator fromEventPattern

`fromEventPattern(addHandler_func: Function): Observable`

- `addHandler_func` - A megadott argumentum `addHandler_func`, ez a tényleges eseményforráshoz lesz csatolva.

```
import { fromEventPattern } from 'rxjs';

function addBtnClickHandler(handler) {
  document.getElementById("btnclick").addEventListener('click', handler);
}

const button_click = fromEventPattern(addBtnClickHandler);
button_click.subscribe(
  x => console.log(
    "ClientX = " + x.clientX + " and ClientY=" + x.clientY
  )
);
```

RxJS - Creation Operator interval

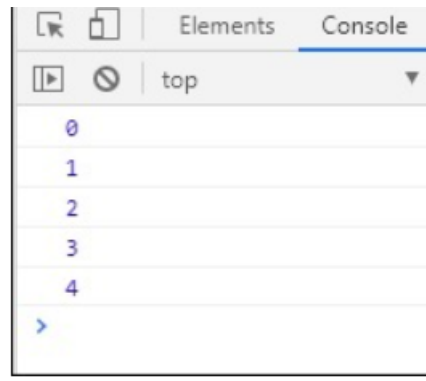
`interval(time): Observable`

- `idő` - A megadott idő ezredmásodpercekben van megadva.

visszatérési érték: observable, amely sorszámot ad a megadott időintervallumhoz.

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

let test = interval(2000);
let case1 = test.pipe(take(5));
case1.subscribe(x => console.log(x));
```



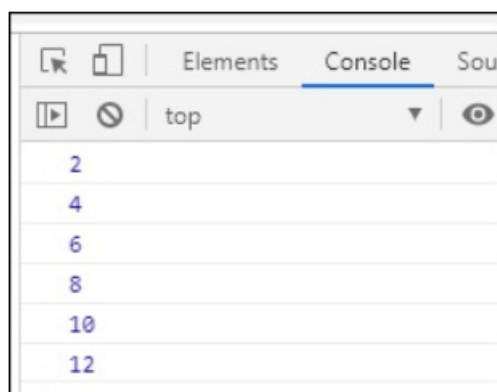
RxJS - Creation Operator of

```
of(input: array[]): Observable
```

- input - A megadott input egy tömb forma.

Observable-t ad vissza a observable forrás értékeivel.

```
import { of } from 'rxjs';
let ints = of(2,4,6,8,10,12);
ints.subscribe(x => console.log(x));
```

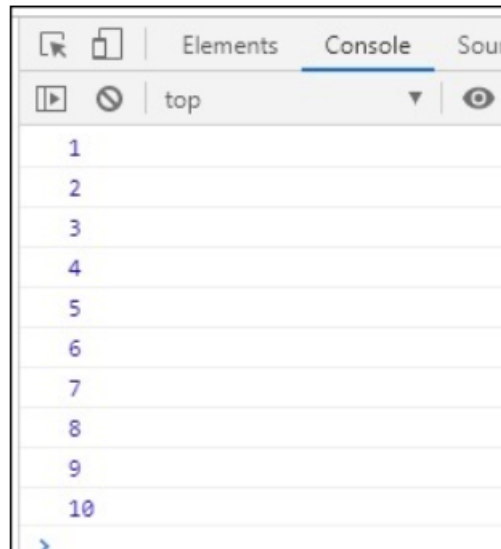


RxJS - Creation Operator range

```
range(start: number, count: number): Observable
```


- start - Az első érték a kezdetektől lesz, és egymás után bocsát ki (emit) a megadott számlálásig.
- count - a kibocsátandó számok száma.

```
import { range } from 'rxjs';
let ints = range(1, 10);
ints.subscribe(x => console.log(x));
```



RxJS - Creation Operator throwError

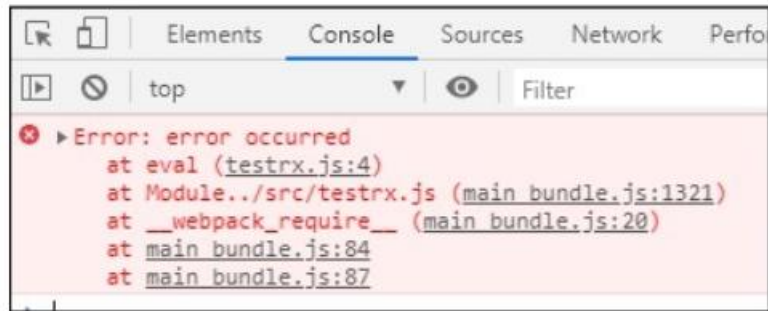
`throwError(error: any): Observable`

- hiba - Az operátor által befogadott érv az a hiba, amelyet értesítenie kell.

Observerbe-t ad vissza, amely értesíti (notify) a hibát.

```
import { throwError, concat, of } from 'rxjs';

const result = throwError(new Error('error occurred'));
result.subscribe(x => console.log(x), e => console.error(e));
```



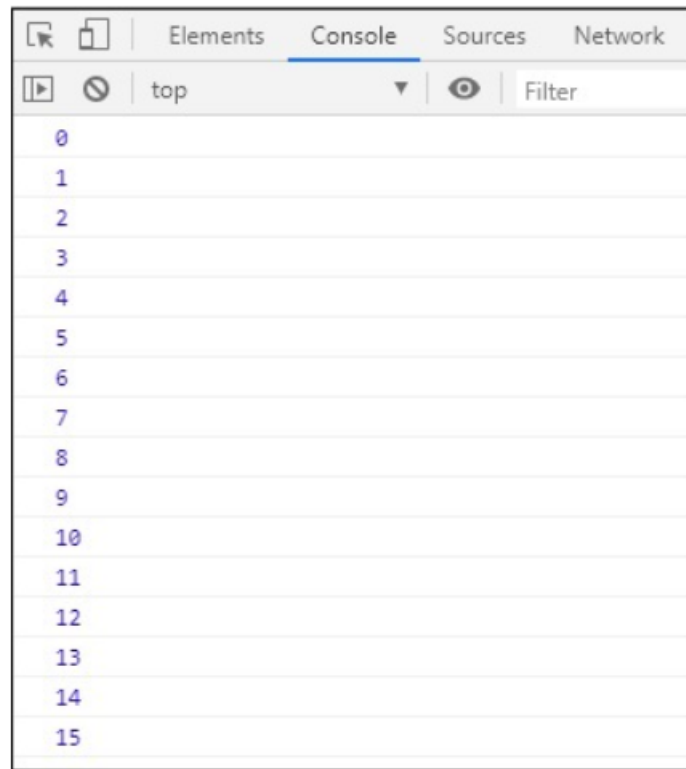
RxJS - Creation Operator timer

`timer(dueTime: number | Date): Observable`

- `dueTime` - milliszekundumban vagy dátumban adható meg.

Observable, amely az időtúllépés után kiadja az értéket, és az érték folyamatosan növekszik minden hívás után.

```
import { timer, range } from 'rxjs';  
  
let all_numbers = timer(10, 10);  
all_numbers.subscribe(x => console.log(x));
```



RxJS - Creation Operator iif

`iif(condition: Function):Observable`

- feltétel - A feltétel függvény, ha a visszatérése igaz, az observable feliratkozik.

Visszatérési érték observable a feltétel alapján.

```
import { iif, of } from 'rxjs';
import { mergeMap, first, last } from 'rxjs/operators';

let task1 = iif(
  () => (Math.random() + 1) % 2 === 0,
  of("Even Case"),
  of("Odd Case")
);
task1.subscribe(value => console.log(value));
```

`iif()` operator olyan mint az inline `if`.

Mathematical Operators

Sr.No	Operator & Description
1	<p>Count</p> <p>A count () operátor paraméterként vár Observable értékeket és átalakítja azt egy Observable-ba, amely egyetlen értéket ad</p>
2	<p>Max</p> <p>A Max módszer minden értéke observable, és a max értékű observable-t adja vissza</p>
3	<p>Min</p> <p>A Min módszer minden értéke observable, és a min értékű observable-t adja vissza.</p>
4	<p>Reduce</p> <p>A reduce operátor esetén az observable bemeneten akkumulátor függvényt használnak, és az akkumulátor függvény visszaküldi a felhalmozott értéket observable formájában, opcionális magértékkal továbbítva az akkumulátor függvénynek.</p> <p>A reduc () függvény 2 argumentumot vesz fel, egy akkumulátorfüggvényt és másodszor a seed-et.</p>

RxJS - Creation Operator count

```
count(predicate_func? : boolean): Observable
```

- predicate_func - (opcionális) Funkció, amely kiszűri a megszámlálandó értékeket az observable forrásból, és logikai értéket ad vissza.

A visszatérő érték observable, amely megadja az adott számok számát.

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';

let all_nums = of(1, 7, 5, 10, 10, 20);
let final_val = all_nums.pipe(count());
final_val.subscribe(x => console.log("The count is "+x));
```

The count is 6

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';
let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(count(a => a % 2 === 0));
final_val.subscribe(x => console.log("The count is "+x));
```

The count is 4

RxJS - Mathematical Operator Max

```
max(comparer_func?: number): Observable
```

- összehasonlító_függvény - - opcionális. Olyan függvény, amely a observable forrásból kiszűri a max értéknek figyelembe veendő értékeket. Ha nincs megadva, akkor az alapértelmezett funkciót vesszük figyelembe.

A visszatérő érték observable, amelynek maximális értéke lesz.

```
import { of } from 'rxjs';
import { max } from 'rxjs/operators';

let all_nums = of(1, 6, 15, 10, 58, 20, 40);
let final_val = all_nums.pipe(max());
final_val.subscribe(x => console.log("The Max value is "+x));
```

The Max value is 58

```
import { from } from 'rxjs';
import { max } from 'rxjs/operators';

let list1 = [1, 6, 15, 10, 58, 2, 40];
let final_val = from(list1).pipe(max((a,b)=>a-b));
final_val.subscribe(x => console.log("The Max value is "+x));
```

The Max value is 58

RxJS - Mathematical Operator Min

```
min(comparer_func?: number): Observable
```

- összehasonlító_függvény - - opcionális. Olyan függvény, amely az observable forrásból kiszűri a min értéknek figyelembe veendő értékeket. Ha nincs megadva, akkor az alapértelmezett függvényt vesszük figyelembe.

A visszatérő érték observable, amelynek min értéke lesz.

```
import { of } from 'rxjs';
import { min } from 'rxjs/operators';

let list1 = [1, 6, 15, 10, 58, 2, 40];
let final_val = of(1, 6, 15, 10, 58, 2, 40).pipe(min());

final_val.subscribe(x => console.log("The Min value is "+x));
```

The Min value is 1

```
import { of ,from} from 'rxjs';
import { min } from 'rxjs/operators';

let list1 = [1, 6, 15, 10, 58, 2, 40];
let final_val = from(list1).pipe(min((a,b) => a - b));
final_val.subscribe(x => console.log("The Min value is "+x));
```

The Min value is 1

RxJS - Mathematical Operator Reduce

reduce(accumulator_func, seeder?) : Observable

- accumulator_func - (opcionális). egy olyan függvény, amelyet az observable elemek forrásértékeire hívnak meg.
- seeder - ((opcionális) Alapértelmezés szerint nincs meghatározva. A felhalmozás szempontjából figyelembe veendő kezdeti érték.

Observable eredményt ad, amelynek egyetlen felhalmozott értéke lesz.

```
import { from } from 'rxjs';
import { reduce } from 'rxjs/operators';

let items = [
  {item1: "A", price: 1000.00},
  {item2: "B", price: 850.00},
  {item2: "C", price: 200.00},
  {item2: "D", price: 150.00}
];
let final_val = from(items).pipe(reduce((acc, itemsdet) => acc+itemsdet.price, 0));
final_val.subscribe(x => console.log("Total Price is: "+x));
```

Total Price is: 2200

Join Operators

Sr.No	Operator & Description
1	<p>concat</p> <p>Ez az operátor egymás után kiadja a bemenetként megadott Observable-t, és folytatja a következőt.</p>
2	<p>forkJoin</p> <p>Ezt az operátort egy tömbben vagy dictionary objektumban vesszük bemenetként, és megvárja, amíg a observable befejezi és visszaadja az adott observable-ből kibocsátott utolsó értékeket.</p>
3	<p>merge</p> <p>Ez az operátor felveszi az observable bemenetet, és az összes értéket kibocsátja az observable értékből, és egyetlen observable kimenetet bocsát ki.</p>
4	<p>race</p> <p>Ez egy observable-t ad vissza, amely az első observable forrás tükörpéldánya lesz.</p>

RxJS - Join Operator concat

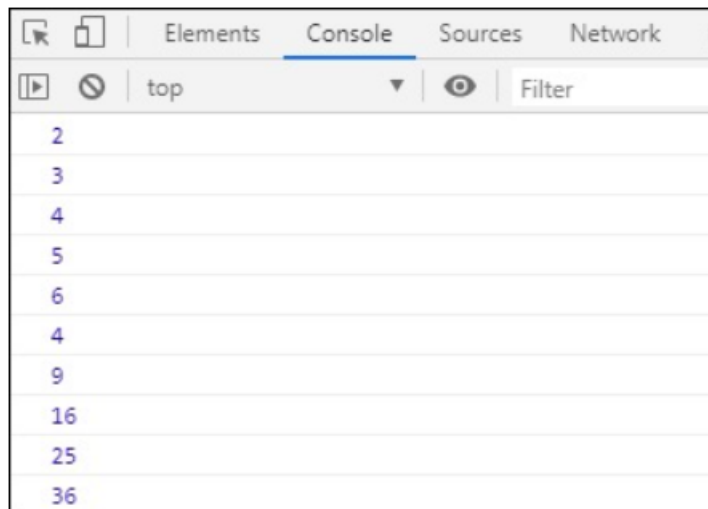
```
concat(observables: Array): Observable
```

- observables- A megadott bemenet egy observable tömb.

Az observable értéket egyetlen értékkel egyesítve adjuk össze az observable forrás értékeivel.

```
import { of } from 'rxjs';
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36);
let final_val = list1.pipe(concat(list2));
final_val.subscribe(x => console.log(x));
```



RxJS - Join Operator forkJoin

`forkJoin(value: array or dict object): Observable`

- érték - Az érték az a bemenet, amely lehet tömb vagy dictionary objektum.

Egy observable értéket az adott observable értékből kibocsátott utolsó értékekkel adunk vissza.

```
import { of, forkJoin } from 'rxjs';
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36);
let final_val = forkJoin([list1, list2]);
final_val.subscribe(x => console.log(x));
```

[6, 36]

RxJS - Join Operator merge

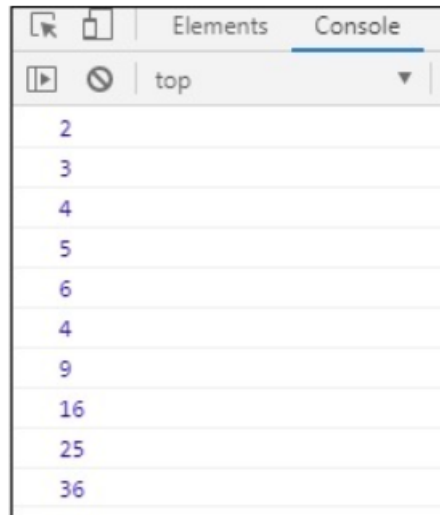
`merge(observable:array[]): Observable`

- observable- A bemenet observable tömb lesz.

Observable eredményt ad, egyetlen kimenetként.


```
import { of, merge } from 'rxjs';
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36);
let final_val = merge(list1, list2);
final_val.subscribe(x => console.log(x));
```



RxJS - Join Operator race

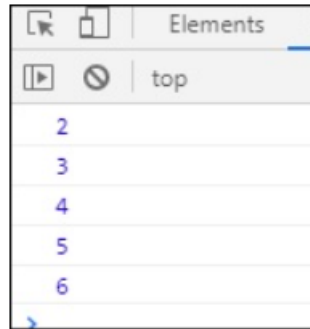
```
race(observables: array[]): Observable
```

- observable- Az operátor argumentuma az observable tömbje.

Observable-t fog visszaadni, amely az első observable forrás tükörpéldánya lesz.

```
import { of, race } from 'rxjs';
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36);
let final_val = race(list1, list2);
final_val.subscribe(x => console.log(x));
```



Transformation Operators

Sr.No	Operator & Description
1	<p>buffer</p> <p>A buffer observable módon működik, és az argumentum observable-ként veszi fel. Elkezd bufferelni a tömbbe eredetileg egy observable értékeket, és ugyanazokat bocsátja ki, amikor az observable argumentumként kibocsátja. Amint az observable kibocsátja, a buffer visszaáll és újra megkezd az eredeti bufferelését, amíg az observable kibocsát és ugyanaz a forgatókönyv megismétlődik.</p>
2	<p>bufferCount</p> <p>A buffercount () operátor esetében összegyűjti az értékeket abból az observable-ből, amelyre hívják, és ugyanazokat bocsátja ki, amikor a bufferszámnak adott bufferméret egyezik.</p>
3	<p>bufferTime</p> <p>Ez hasonló a bufferCount-hoz, így itt összegyűjti az értékeket abból az observable-ből, amelyre meghívják, és kiadja a bufferTimeSpan elemet. 1 argumentumot vesz igénybe, azaz bufferTimeSpan.</p>
4	<p>bufferToggle</p> <p>A bufferToggle () esetében 2 argumentumra van szükség, opening és closingSelector. A nyitó argumentumok előfizethetők (subscriber), vagy promise a buffer elindítására, a második argumentum pedig closingSelector, vagy promise-nak egy mutatót a puffer bezárására és az összegyűjtött értékek kibocsátására.</p>
5	<p>bufferWhen</p> <p>Ez az operátor megadja az értékeket a tömb formában, egy argumentumot vesz fel egy függvényként, amely eldönti, hogy mikor kell bezárni (close), kibocsátani (emit) és visszaállítani (reset) a puffert.</p>
6	<p>expand</p> <p>Az operátor egy függvényt vesz fel argumentumként, amelyet a rekurzívan observable</p>

	forráson és az observable kimeneten alkalmaznak. A végső érték observable.
7	groupBy A groupBy operátorban a kimenetet egy adott feltétel alapján csoportosítják, és ezeket a csoportelemeket GroupedObservable néven bocsátják ki.(emit)
8	map Map operátor esetén egy projektfüggvényt alkalmaznak az Observable forrás minden értékére, és ugyanazt a kimenetet bocsátják ki (emit), mint az Observable forrást.
9	mapTo Egy konstans értéket adunk meg kimenetként az Observable-el együtt, valahányszor a Observable forrás értéket bocsát ki.
10	mergeMap A mergeMap operátor esetében egy projekt függvényt alkalmaznak minden egyes forrásértékre, és ennek kimenetét összevonják az Observable kimenettel.
11	switchMap A switchMap operátor esetében egy projektfüggvényt alkalmaznak minden forrásértékre, és ennek kimenetét összevonják az Observable kimenettel, és a megadott érték a legutóbb vetített Observable.
12	window Megkövetel egy argumentum windowboundaries, amelyek observable-ek és visszaadnak egy beágyazott observable-t, amikor az adott windowboundaris kiadnak

RxJS - Transformation Operator buffer

`buffer(input_observable: Observable): Observable`

- `input_observable` - observable, amely a puffert értékeket bocsátja ki. Például gomra kattintás.

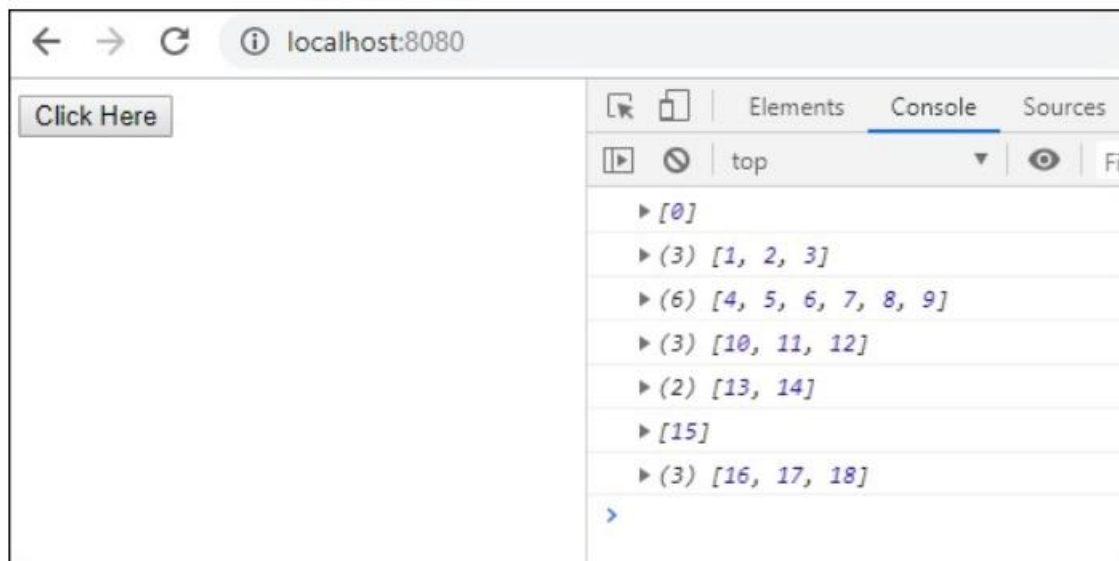
Egy observable eredményt adunk vissza, amely puffert értékeket tartalmaz.

```
import { fromEvent, interval } from 'rxjs';
import { buffer } from 'rxjs/operators';

let btn = document.getElementById("btnclick");

let btn_clicks = fromEvent(btn, 'click');
let interval_events = interval(1000);
let buffered_array = interval_events.pipe(buffer(btn_clicks));
```

```
buffered_array.subscribe(arr => console.log(arr));
```



RxJS - Transformation Operator bufferCount

```
bufferCount(bufferSize: number, startBufferEvery: number = null): Observable
```

- bufferSize - A kibocsátandó buffer mérete.

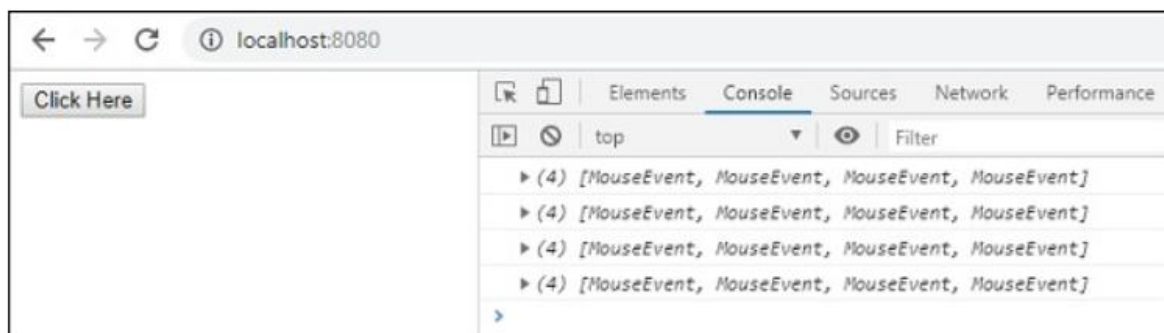
Egy observable eredményt adunk vissza, amely bufferelt értékeket tartalmaz.

```
import { fromEvent } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

let btn = document.getElementById("btnclick");

let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferCount(4));
buffered_array.subscribe(arr => console.log(arr));
```

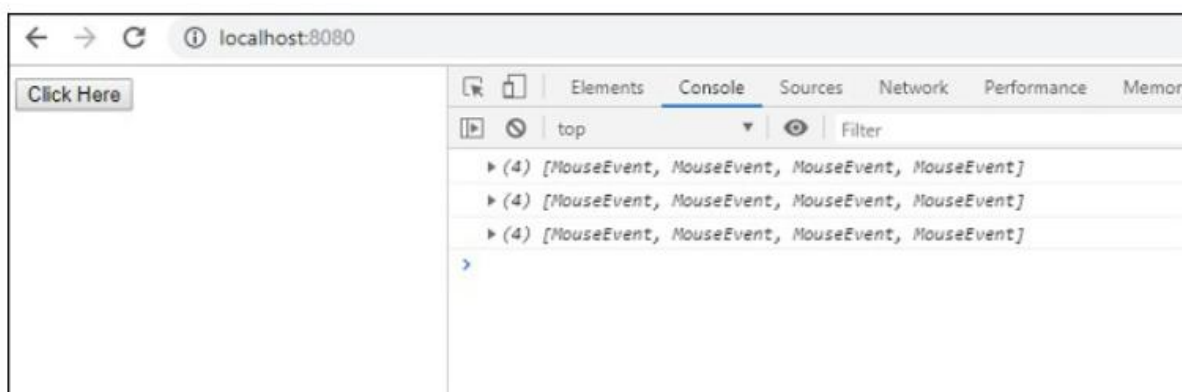
A fenti példában a bufferSize értéke 4. Tehát 4 kattintás számlálása után a kattintási események tömbje összegyűlik egy tömbben, és megjelenik. Mivel nem adtuk meg a startBufferEvery értékeket, az értékeket a kezdetektől számoljuk.



```
import { fromEvent } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferCount(4, 2));
buffered_array.subscribe(arr => console.log(arr));
```

Ebben a példában hozzáadtuk a `startBufferEvery`-t, így minden 2 kattintás után 4 kattintási esemény pufforszámát jeleníti meg.



RxJS - Transformation Operator `bufferTime`

`bufferTime(bufferTimeSpan: number): Observable`

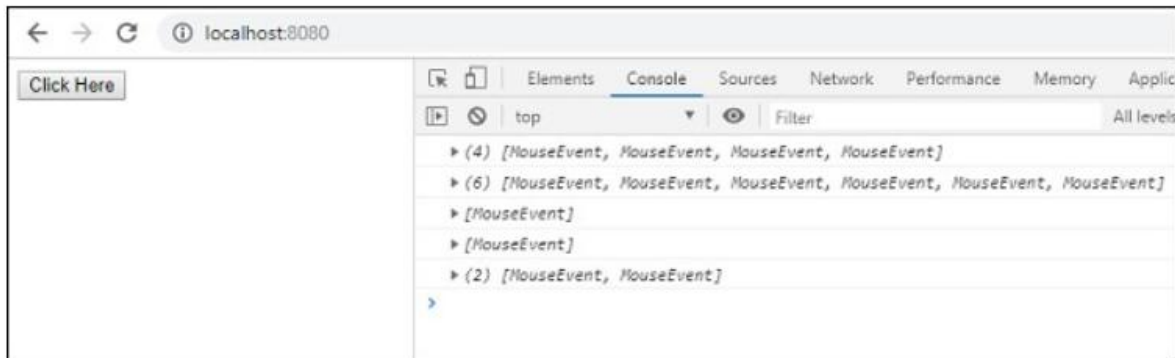
- `bufferTimeSpan` - A puffertömb kitöltésének ideje.

Egy observable eredményt adunk vissza, amely puffereelt értékeket tartalmaz.

```
import { fromEvent } from 'rxjs';
import { bufferTime } from 'rxjs/operators';
```

```
let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferTime(4000));
buffered_array.subscribe(arr => console.log(arr));
```

A példában az alkalmazott idő 4 másodperc, tehát a `bufferTime()` operátor összegyűjti a kattintásokat, és 4 másodpercenként megjeleníti őket az alábbiak szerint.



RxJS - Transformation Operator bufferToggle

```
bufferToggle(openings: SubscribableOrPromise, closingSelector:
SubscribableOrPromise): Observable
```

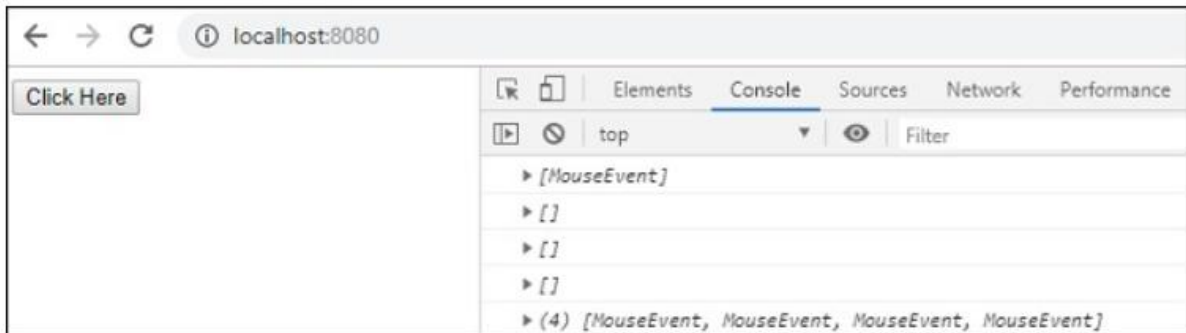
- `openings` - Promise vagy notification az új puffer elindítására.
- `closingSelector` - Olyan függvény, amely az `openings` observable értékeit veszi fel, és visszatér az `Subscribable` vagy `Promise`-al.

Egy observable eredményt adunk vissza, amely puffertelt értékeket tartalmaz.

```
import { fromEvent, interval, EMPTY } from 'rxjs';
import { bufferToggle } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let start = interval(2000);
let buffered_array = btn_clicks.pipe(
  bufferToggle(start, a => a%2 == 0 ? interval(1000): EMPTY)
);
buffered_array.subscribe(arr => console.log(arr));
```

A fenti példában a puffer 2 másodperc múlva kezdődik és akkor ér véget, amikor 1-es időközönként intervallumot kapunk, ha a kapott érték még ellenkező esetben kiüríti a pufferértékeket és üres értékeket bocsát ki.



RxJS - Transformation Operator bufferWhen

`bufferWhen(closing_func: Observable): Observable`

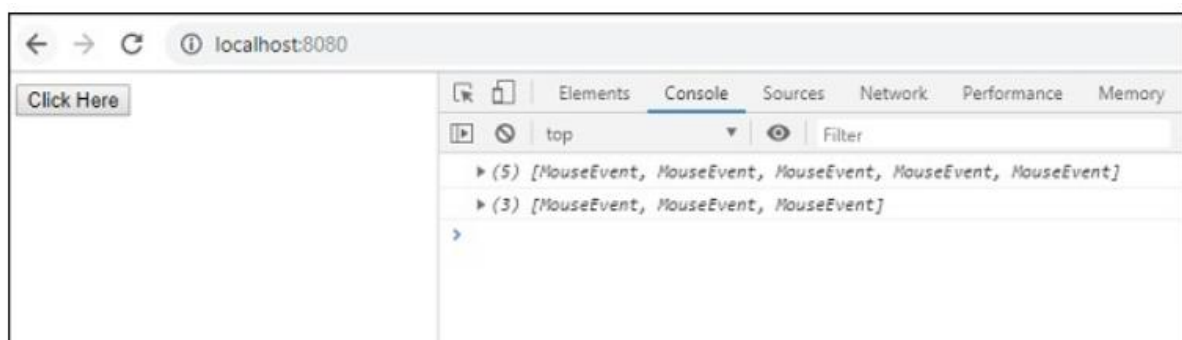
- `closing_func` – Observable jelző pufferzárást eredményező függvény.

Egy observable eredményt adunk vissza, amely pufferelt értékeket tartalmaz.

```
import { fromEvent, interval } from 'rxjs';
import { bufferWhen } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferWhen(() => interval(5000)));
buffered_array.subscribe(arr => console.log(arr));
```

Amikor megadtunk egy olyan függvényt, amely 5 másodpercenként hajt végre. Tehát minden 5 másodperc után kiadja az összes rögzített kattintást, visszaállítja és újraindítja.



RxJS - Transformation Operator expand

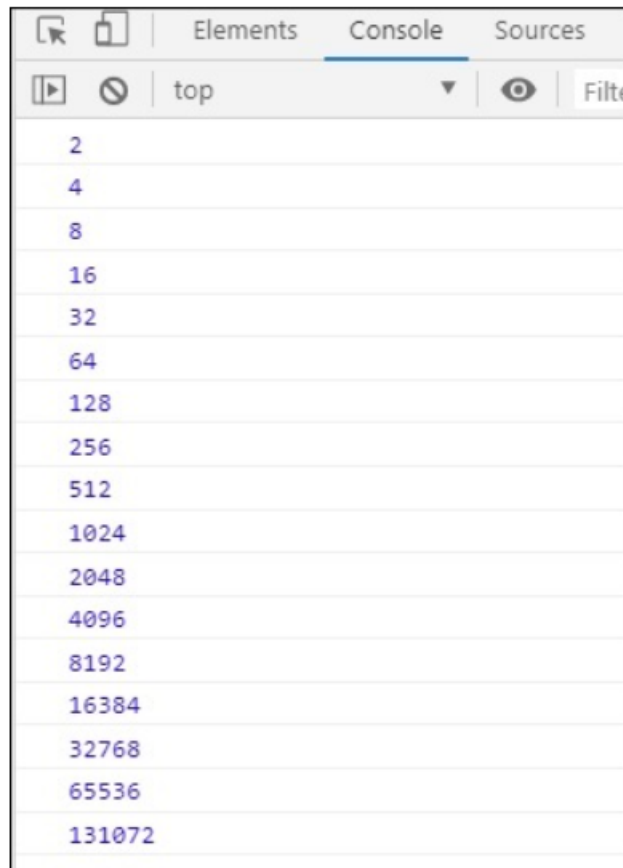
`expand(recursive_func:observable): Observable`

- `recursive_func` - A függvényt a forrásból származó összes értékre alkalmazzák, és Observable-t ad vissza.

Visszatérési érték: observable, a `recursive_func` eredményének megfelelő értékekkel.

```
import { of } from 'rxjs';
import { expand } from 'rxjs/operators';

let buffered_array = of(2).pipe(expand(x => of(2 * x)));
buffered_array.subscribe(arr => console.log(arr));
```



RxJS - Transformation Operator `groupBy`

```
groupBy(keySelector_func: (value: T) => K):GroupedObservables
```

- `keySelector_func` - Olyan függvény, amely megadja a kulcsot a megfigyelhető forrás minden eleméhez.

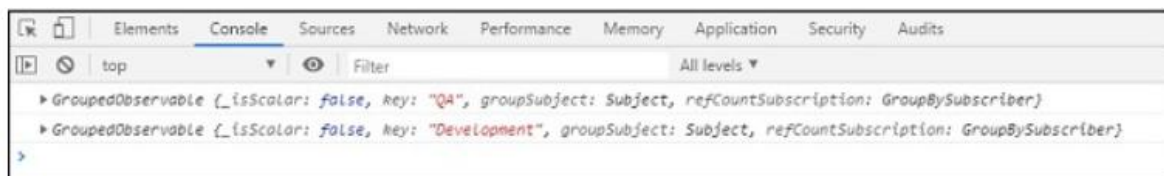
A visszatérési érték egy Observable, amely `GroupedObservables` néven ad ki értékeket.


```
import { of, from } from 'rxjs';
import { groupBy } from 'rxjs/operators';

const data = [
  {groupId: "QA", value: 1},
  {groupId: "Development", value: 3},
  {groupId: "QA", value: 5},
  {groupId: "Development", value: 6},
  {groupId: "QA", value: 2},
];

from(data).pipe(
  groupBy(item => item.groupId)
)
.subscribe(x => console.log(x));
```

Ha látja a kimenetet, akkor observable, ahol az elemek csoportosítva vannak. Az általunk megadott adatoknak két csoportja van: minőségbiztosítás és fejlesztés. A kimenet az alábbiak csoportosítását mutatja –



RxJS - Transformation Operator map

`map(project_func: function): Observable`

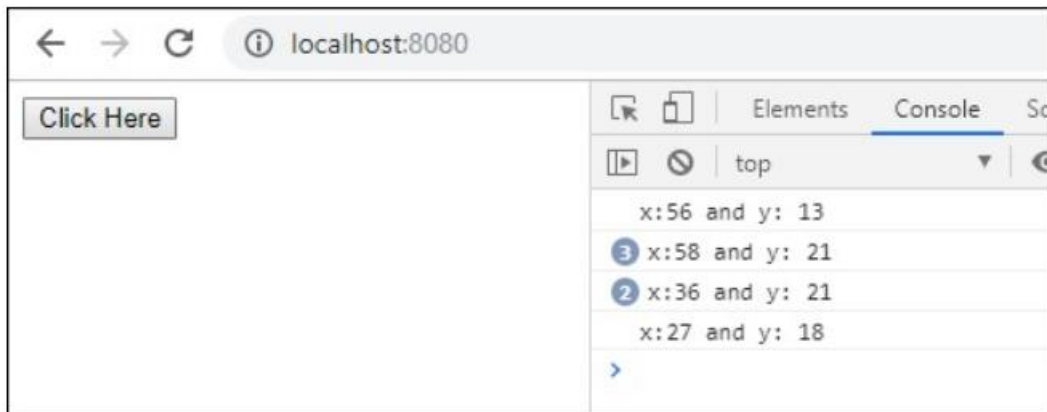
- `project_func` - A `project_func` argumentumot veszi fel, amelyet a forrás összes observable értékére alkalmazunk.

Observable, a `project_func` eredményének megfelelő értékekkel.

```
import { fromEvent } from 'rxjs';
import { map } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');

let positions = btn_clicks.pipe(map(ev => ev));
positions.subscribe(x => console.log("x:" + x.clientX + " and y: " + x.clientY));
```



RxJS - Transformation Operator mapTo

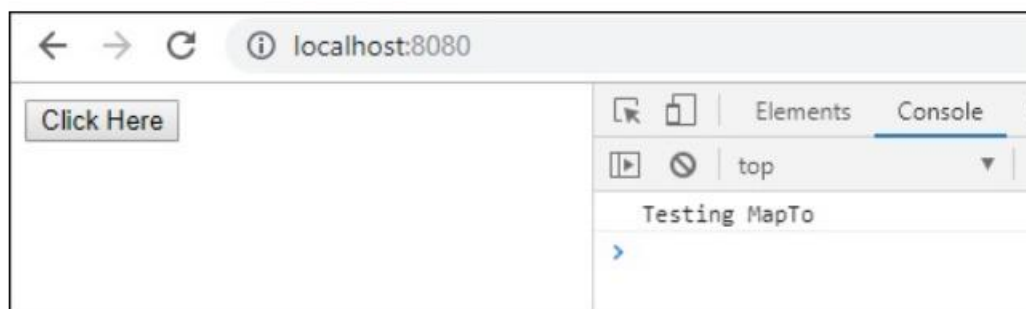
`mapTo(value: any): Observable`

- érték - Az értéket argumentumként veszi fel, és ez az érték a megadott forrásértéket képezi.

Observable értéket ad vissza, ha a forrás observable értéket kibocsát.

```
import { fromEvent } from 'rxjs';
import { mapTo } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let positions = btn_clicks.pipe(mapTo ("Testing MapTo"));
positions.subscribe(x => console.log(x));
```



RxJS - Transformation Operator mergeMap

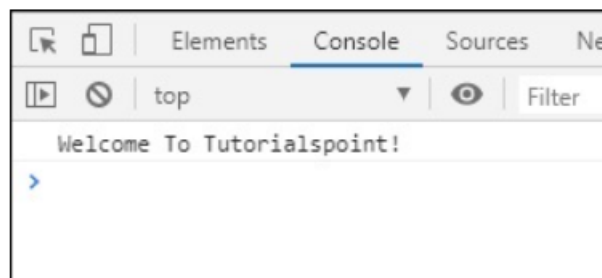
`mergeMap(project_func: function): Observable`

- `project_func` - A `project_func` argumentumot veszi fel, amelyet a forrás összes observable értékére alkalmazunk.

Observable értéket ad vissza, amelynek értékei a `project_func` alapján alapulnak a megfigyelhető források minden értékén.

```
import { of } from 'rxjs';
import { mergeMap, map } from 'rxjs/operators';

let text = of('Welcome To');
let case1 = text.pipe(mergeMap((value) => of(value + ' Tutorialspoint!')));
case1.subscribe((value) => {console.log(value)});
```



RxJS - Transformation Operator `switchMap`

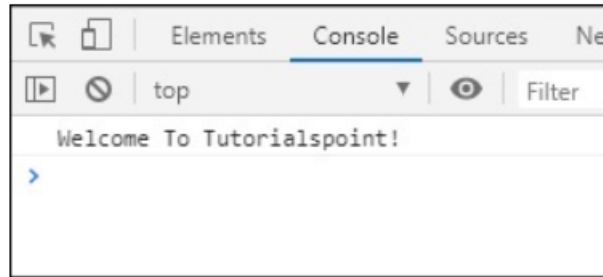
`switchMap(project_func: function): Observable`

- `project_func` - A `project_func` argumentumot veszi fel, amelyet az observable forrás összes kibocsátott értékére alkalmazunk, és egy Observable értéket ad vissza.

A visszatérési érték egy observable, amelynek a `project_func` alapú értékei vannak a forrás minden observable értékére alkalmazva.

```
import { of } from 'rxjs';
import { switchMap } from 'rxjs/operators';

let text = of('Welcome To');
let case1 = text.pipe(switchMap((value) => of(value + ' Tutorialspoint!')));
case1.subscribe((value) => {console.log(value)});
```



RxJS - Transformation Operator window

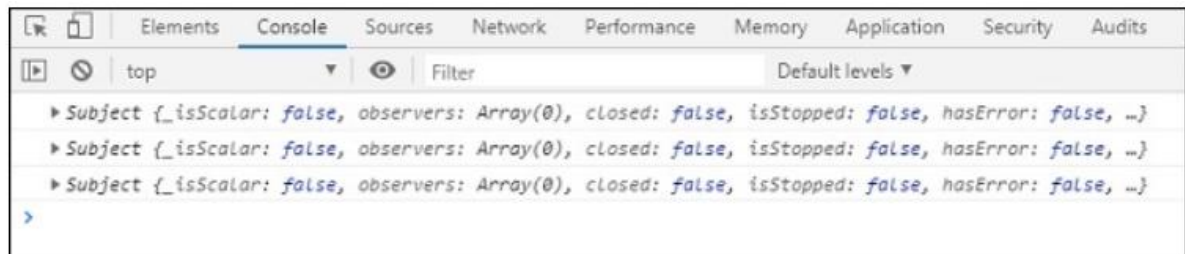
`window(windowBoundaries: Observable): Observable`

- `windowBoundaries` - Az argumentum `windowBoundaries` egy observable.

Megfigyelhető ablakokat (windows) ad vissza.

```
import { fromEvent, interval } from 'rxjs';
import { window } from 'rxjs/operators';

let btnclick = fromEvent(document, 'click');
let sec = interval(5000);
let result = btnclick.pipe(window(interval(5000)));
result.subscribe(x => console.log(x));
```



Filtering Operators

Sr.No	Operator & Description
1	<p>debounce</p> <p>A forrásból kibocsátott érték egy idő után observable, és a kibocsátást egy másik observable vagy promise bemenet határozza meg.</p>
2	debounceTime

	Csak az idő lejártá után bocsát ki értéket az observable forrásból.
3	distinct Ez az operátor megadja a forrás összes observable értékét, amelyek különböznek az előző értékhez képest.
4	elementAt Ez az operátor egyetlen értéket ad az observable forrásból a megadott index alapján.
5	filter Ez az operátor a megadott predikátumfüggvény alapján kiszűri az Observable forrás értékeit.
6	first Ez az operátor megadja az Observable forrás által kibocsátott első értéket.
7	last Ez az operátor megadja az Observable forrás által kibocsátott utolsó értéket.
8	ignoreElements Ez az operátor figyelmen kívül hagyja az observable forrás összes értékét, és csak a callback hívásokat (sikeres vagy hibás) hajtja végre.
9	sample Ez az operátor megadja az observable forrás legfrissebb értékét, és a kimenet a neki átadott argumentumtól függ.
10	skip Ez az operátor visszaküld egy observable információt, amely kihagyja a bemenetként vett számolási elemek első előfordulását.
11	throttle Ez az operátor az argumentumként vett bemeneti függvény által meghatározott ideig kimeneti és figyelmen kívül hagyja a forrás értékeit, és ugyanaz a folyamat megismétlődik.

RxJS - Filtering Operator debounce

```
debounce(durationSelector: Observable or promise): Observable
```

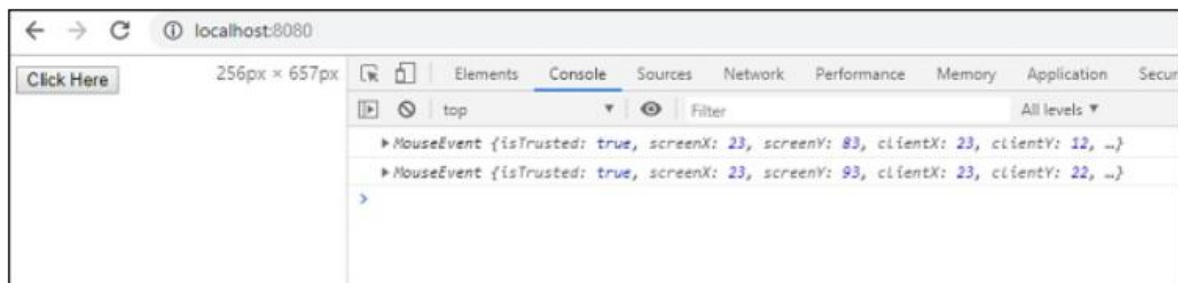
- durationSelector - Olyan argumentumot vesz be, amelyet úgy hívnak: DurationSelector, amely observable-t vagy promise-t ad. Ez az argumentum az observable forrásból kap inputot, és eldönti az egyes forrásértékek időtűllépését.

Observable értéket ad vissza, ahol az observable forrás kibocsátása késleltetődik az durationSelector alapján.

```
import { fromEvent, interval } from 'rxjs';
import { debounce } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(debounce(() => interval(2000)));
case1.subscribe(x => console.log(x));
```

Itt a debounce () operátor segítségével késik a kattintási esemény



RxJS - Filtering Operator debounceTime

debounceTime(dueTime: number): Observable

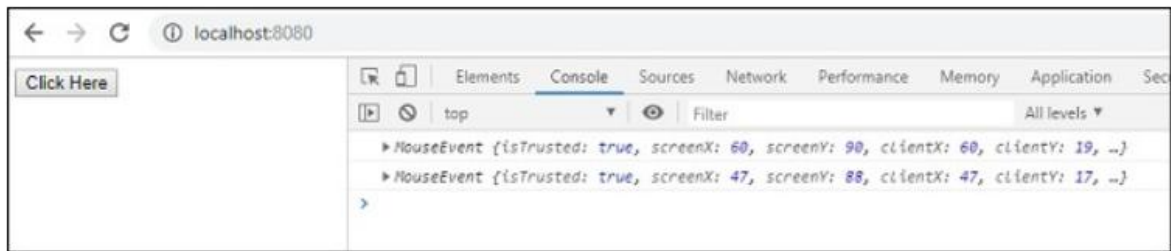
- debounceTime - A dueTime argumentum ezredmásodpercekben megadott időtűllépés.

Observable értéket ad vissza, ahol az observable forrás kibocsátása késik a dueTime alapján.

```
import { fromEvent } from 'rxjs';
import { debounceTime } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(debounceTime(2000));
case1.subscribe(x => console.log(x));
```

Ugyanaz, mint a debounce () operátor, azzal a különbséggel, hogy a késleltetési időt közvetlenül ennek az operátornak adhatja át.



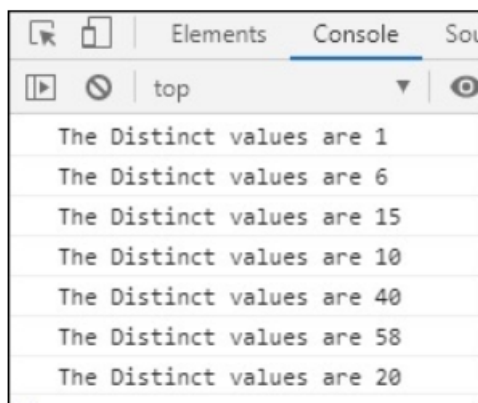
RxJS - Filtering Operator distinct

`distinct()`

Observable eredményt ad, amelynek különböző értékei vannak.

```
import { of } from 'rxjs';
import { distinct } from 'rxjs/operators';

let all_nums = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40);
let final_val = all_nums.pipe(distinct());
final_val.subscribe(x => console.log("The Distinct values are "+x));
```



RxJS - Filtering Operator elementAt

`elementAt(index: number): Observable`

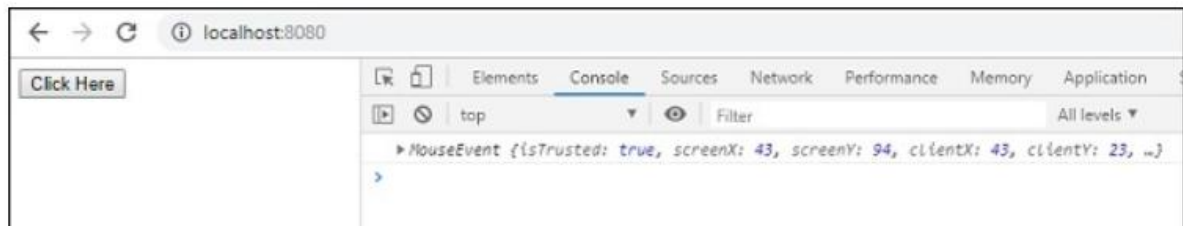
- `index` - Az átadott argumentum a típuszám indexe, 0-tól kezdődően. Az indexhez observable forrás értékét visszaadjuk.

Egy observable értéket, amely a megadott indexen alapul, ad vissza.

```
import { fromEvent } from 'rxjs';
import { elementAt } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(elementAt(4));
case1.subscribe(x => console.log(x));
```

Az `elementAt (4)` elemet használtuk, így az 5. kattintás adódik, mivel az index 0-tól kezdődik.



RxJS - Filtering Operator filter

```
filter(predicate_func: function): Observable
```

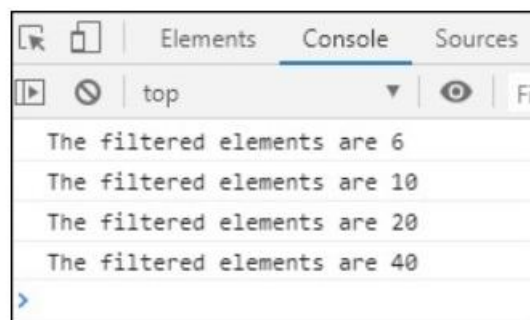
- `predicate_func` - A `predicate_func` egy logikai értéket ad vissza, és a kimenet szűrésre kerül, ha a függvény igaz értéket ad vissza.

Observable értékeket ad vissza, amelyek kielégítik a `predicate_func` predikátumot.

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(filter(a => a % 2 === 0));
final_val.subscribe(x => console.log("The filtered elements are "+x));
```

A páros számokat a `filter ()` operátor segítségével szűrtük.



RxJS - Filtering Operator first

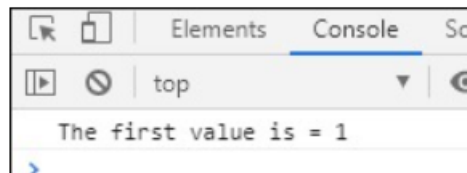
`first()`

Observable értéket adunk vissza, az elsőt.

```
import { of } from 'rxjs';
import { first } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(first());
final_val.subscribe(x => console.log("The first value is = "+x));
```

A `first ()` operátor megadja az első értéket a megadott listából.



RxJS - Filtering Operator last

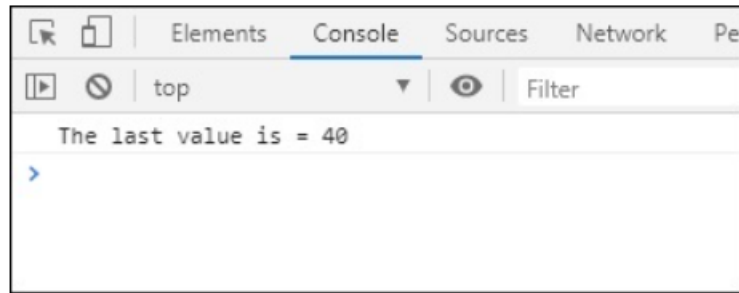
`last()`

Observable ad vissza az utolsó értékkel.

```
import { of } from 'rxjs';
import { last } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(last());
final_val.subscribe(x => console.log("The last value is = "+x));
```

A `last ()` adja meg az utolsó értéket a megadott listából.



RxJS - Filtering Operator ignoreElements

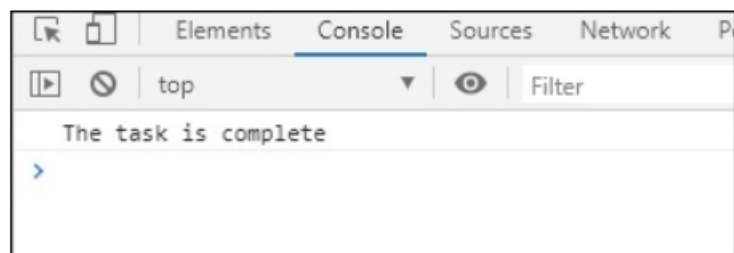
`ignoreElements()`

Observable eredményt ad, amely a forrás alapján complete vagy error-t hívja.

```
import { of } from 'rxjs';
import { ignoreElements } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(ignoreElements());
final_val.subscribe(
  x => console.log("The last value is = "+x),
  e => console.log('error:', e),
  () => console.log('The task is complete')
);
```

Az `ignoreElements()` operátor közvetlenül végrehajtja a teljes metódust, ha siker és hiba, ha nem, és figyelmen kívül hagy minden mást.



RxJS - Filtering Operator sample

`sample(notifier: Observable): Observable`

- **notifier** - Az argumentum notifier egy observable, amely eldönti azt a kimenetet, amelyet kell választani.

Observables-t ad vissza, a forrás kiad.

```
import { fromEvent, interval } from 'rxjs';
import { sample } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(sample(interval(4000)));
case1.subscribe(x => console.log(x));
```

A `sample()` operátor megkapja az `interval(4000)`-t, így a kattintási esemény akkor jelenik meg, ha a 4 másodperces intervallum megtörtént.



RxJS - Filtering Operator skip

`skip(count: number): Observable`

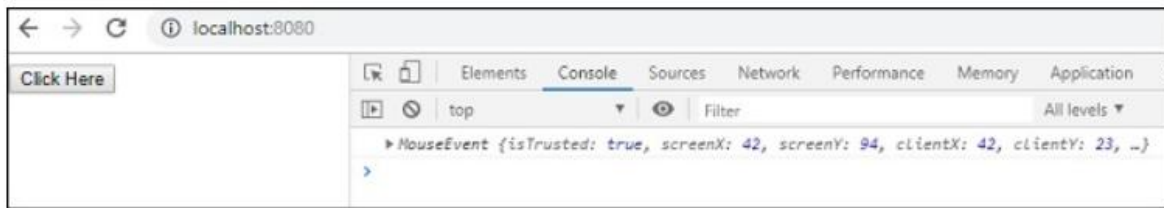
- **count** - Az argumentumszám az, hogy hányszor hagyják ki az elemeket az observable forrásból.

Observable eredményt ad vissza, amely az adott szám alapján kihagyja az értékeket.

```
import { fromEvent, interval } from 'rxjs';
import { skip } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(skip(2));
case1.subscribe(x => console.log(x));
```

A `skip()` operátornak 2-t adunk, így az első két kattintást figyelmen kívül hagyjuk, és a harmadik kattintási esemény emittálásra kerül.



RxJS - Filtering Operator throttle

`throttle(durationSelector: Observable or Promise): Observable`

- **durationSelector** - Az `durationSelector` argumentum egy observable vagy promise, amely figyelmen kívül hagyja az observable forrásból kibocsátott értékek értékeit.

Megfigyelhető értéket ad vissza, amely throttle a megfigyelhető forrásból kibocsátott értékeket.

```
import { fromEvent, interval } from 'rxjs';
import { throttle } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(throttle(ev => interval(2000)));
case1.subscribe(x => console.log(x));
```

Ha rákattint a gombra, az első kattintási esemény kibocsátásra kerül, a későbbi kattintások késleltetésre kerülnek a throttle () kezelőjének adott időtartamig.



Utility Operators

Sr.No	Operator & Description
1	<p>tap</p> <p>Ennek az operátornak meg lesz a kimenete, amely megegyezik az observable forrásával, és felhasználható az értékek megfigyelésre a felhasználó számára történő naplózásához. A fő érték, hiba, ha van, ha a feladat kész.</p>

2	<p>delay</p> <p>Ez az operátor késlelteti az observable forrásból kibocsátott értékeket a megadott időtúllépés alapján.</p>
3	<p>delayWhen</p> <p>Ez az operátor késlelteti az observable forrásból kibocsátott értékeket egy másik, bemenetként vett observable időkorlátja alapján.</p>
4	<p>observeOn</p> <p>Ez a bemeneti ütemezőn alapuló operátor újból elküldi az Observable forrás értesítéseit.</p>
5	<p>subscribeOn</p> <p>Ez az operátor segít aszinkron módon feliratkozni az observable forrásra a bemenetnek vett ütemező alapján.</p>
6	<p>timeInterval</p> <p>Ez az operátor visszaad egy objektumot, amely tartalmazza az aktuális értéket és az aktuális és az előző érték között eltelt időt, amelyet az ütemező bemenetével számítottak ki.</p>
7	<p>timestamp</p> <p>Visszaadja az időbélyeget az Observable forrásból kibocsátott értékkel együtt, amely az érték kibocsátásának idejéről szól.</p>
8	<p>timeout</p> <p>Ez az operátor hibát dob, ha az observable forrás nem ad ki értéket a megadott időtúllépés után.</p>
9	<p>toArray</p> <p>Összegzi az összes forrásértéket az Observable-ból és tömbként adja ki, amikor a forrás befejeződik.</p>

RxJS - Utility Operator tap

```
tap(observer, error, complete):Observable
```

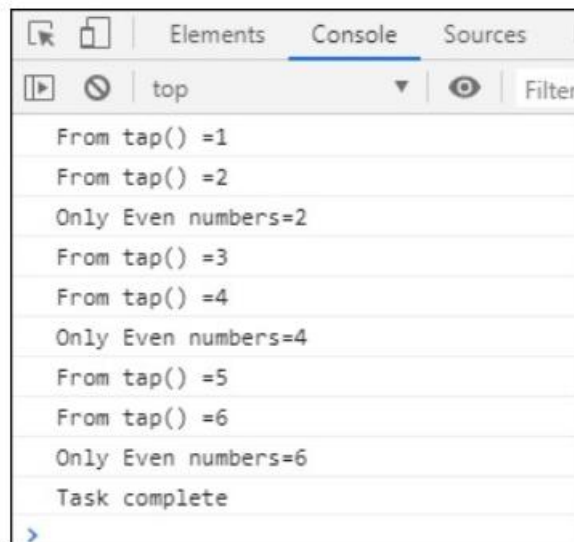
- observer - (nem kötelező) ez ugyanaz, mint az observable forrás.
- error - (opcionális) hibametódus, ha bármilyen hiba lép fel.
- complete - (opcionális) complete () metódust a feladat befejezése után hívják meg.

Observable-el tér vissza

```
import { of } from 'rxjs';
```

```
import { tap, filter } from 'rxjs/operators';

let list1 = of(1, 2, 3, 4, 5, 6);
let final_val = list1.pipe(
  tap(x => console.log("From tap() =" + x),
    e => console.log(e),
    () => console.log("Task complete")),
  filter(a => a % 2 === 0)
);
final_val.subscribe(x => console.log("Only Even numbers=" + x));
```



RxJS - Utility Operator delay

`delay(timeout: number): Observable`

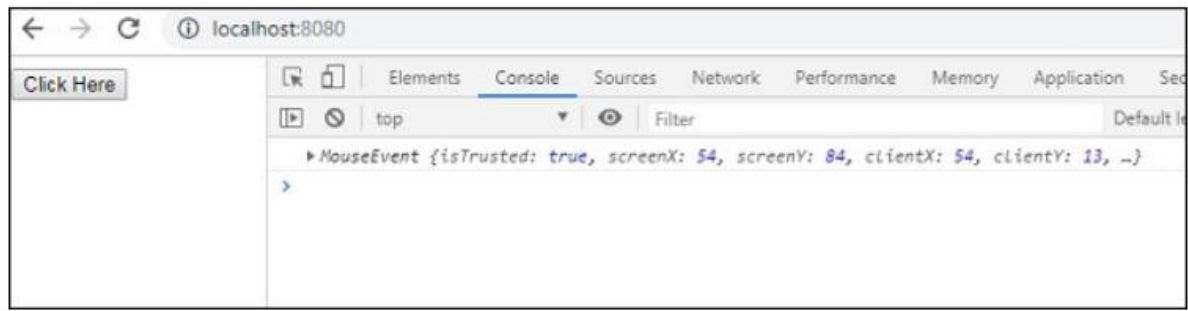
- időtúllépés - Ez milliszekundumokban vagy egy Dátumban lesz, amely késlelteti az értékek kibocsátását a forrásból.

Visszaad egy observable elemet, amely a megadott időtúllépés vagy dátum alapján késlelteti az observable forrást.

```
import { fromEvent } from 'rxjs';
import { delay } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(delay(2000));
case1.subscribe(x => console.log(x));
```

Itt a `delay ()` operátor segítségével késik a kattintási esemény



RxJS - Utility Operator delayWhen

`delayWhen(timeoutSelector_func: Observable): Observable`

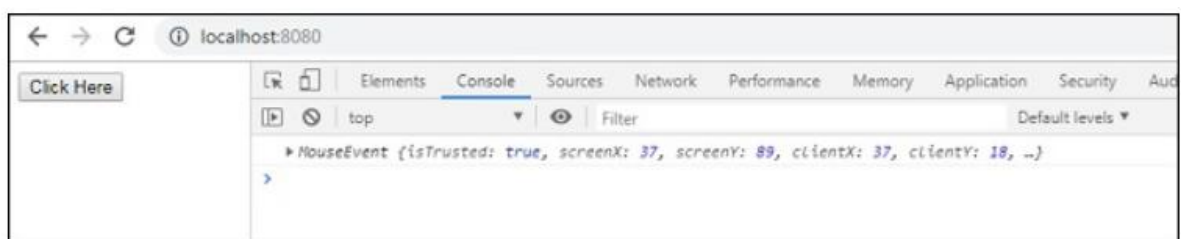
- `timeoutSelector_func` - observable, amely dönt az időtúllépésről.

Observable lesz, amely a `timeoutSelector_func` kimenetet használja az observable forrás késleltetésére.

```
import { fromEvent, timer } from 'rxjs';
import { delayWhen } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(delayWhen(() => timer(1000)));
case1.subscribe(x => console.log(x));
```

Observable-t használtunk a `delayWhen()` használatához, és amikor ez az observable, a kattintási esemény kibocsátásra kerül.



RxJS - Utility Operator observeOn

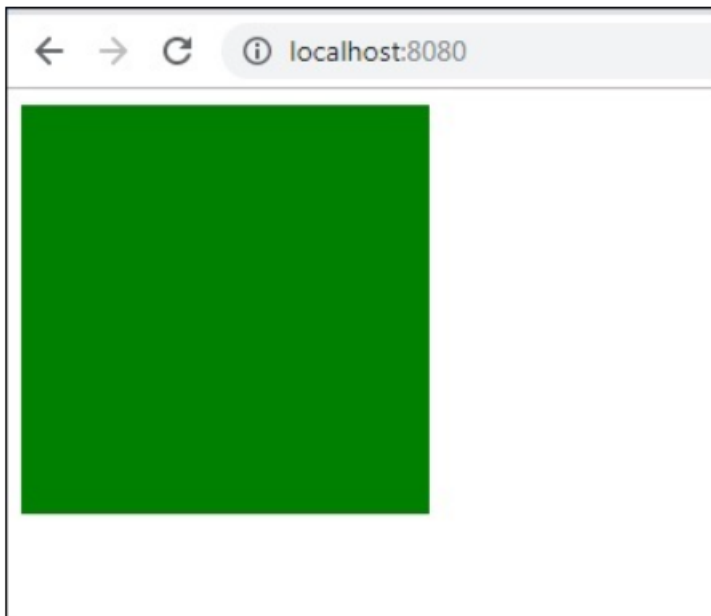
`observeOn(scheduler): Observable`

- **ütemező** - Az ütemezőt bemenetként használják, amely segít az értesítések újbóli kiadásában az observable forrásból.

Observable azonos lesz, mint a forrás observable, de ütemező paraméter-rel.

```
import { interval } from 'rxjs';
import { observeOn } from 'rxjs/operators';
import { animationFrameScheduler } from 'rxjs';

let testDiv = document.getElementById("test");
const intervals = interval(100);
let case1 = intervals.pipe(
  observeOn(animationFrameScheduler),
);
let sub1 = case1.subscribe(val => {
  console.log(val);
  testDiv.style.height = val + 'px';
  testDiv.style.width = val + 'px';
});
```



RxJS - Utility Operator subscribeOn

`subscribeOn(scheduler): Observable`

- ütemező - Az ütemezőt bemenetként használják, amely segít az értesítések újbóli kiadásában az observable forrásból.

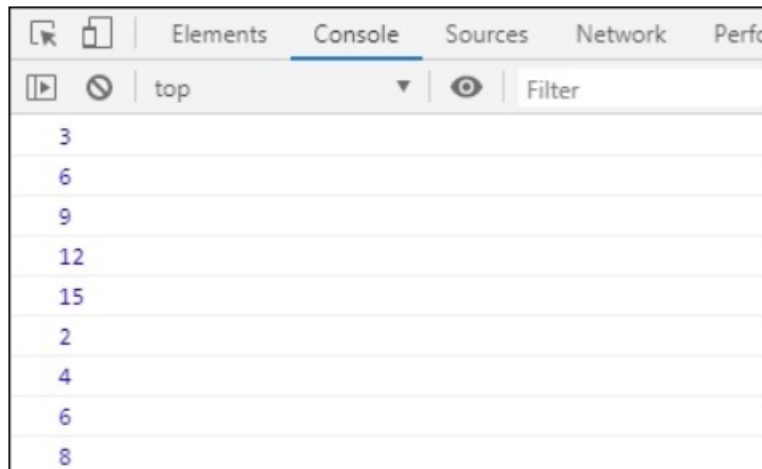
Observable azonos lesz, mint a forrás observable, de ütemező paraméter-rel.

```
import { of, merge, asyncScheduler } from 'rxjs';
import { subscribeOn } from 'rxjs/operators';

let test1 = of(2, 4, 6, 8).pipe(subscribeOn(asyncScheduler));
let test2 = of(3, 6, 9, 12, 15);
```



```
let sub1 = merge(test1, test2).subscribe(console.log);
```



RxJS - Utility Operator `timeInterval`

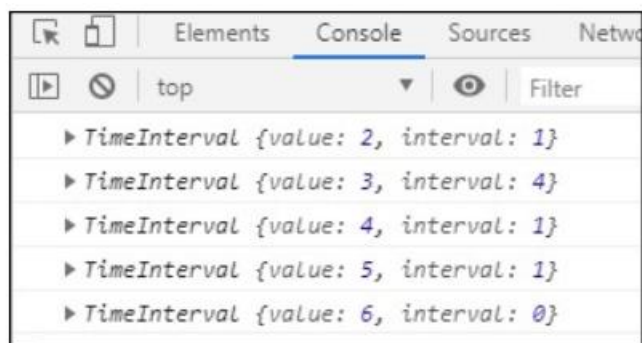
`timeInterval(scheduler): Observable`

- **ütemező** - (opcionális) Az ütemező bemenetével kiszámolható az aktuális és az előző érték között eltelt idő az observable forrásból.

Observable értéket ad vissza, amelynek forrásértékei és időintervalluma is lesz.

```
import { of } from 'rxjs';
import { filter, timeInterval } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let final_val = list1.pipe(timeInterval());
final_val.subscribe(x => console.log(x));
```



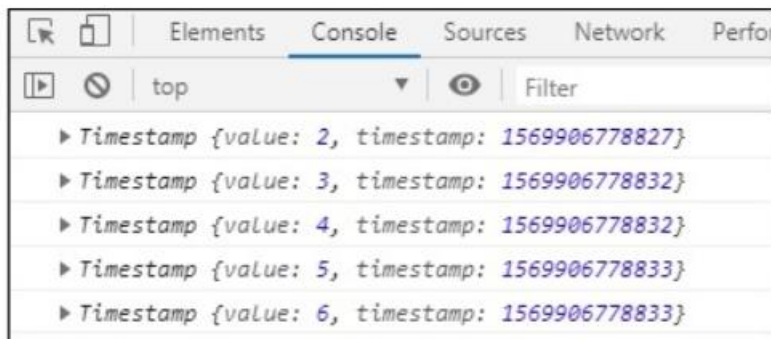
RxJS - Utility Operator timestamp

`timestamp(): Observable`

Visszaadja az időbélyeget az Observable forrásból kibocsátott értékkel együtt, amely az érték kibocsátásának idejéről szól.

```
import { of } from 'rxjs';
import { filter, timestamp } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let final_val = list1.pipe(timestamp());
final_val.subscribe(x => console.log(x));
```



RxJS - Utility Operator timeout

`timeout(timeout: number | Date): Observable`

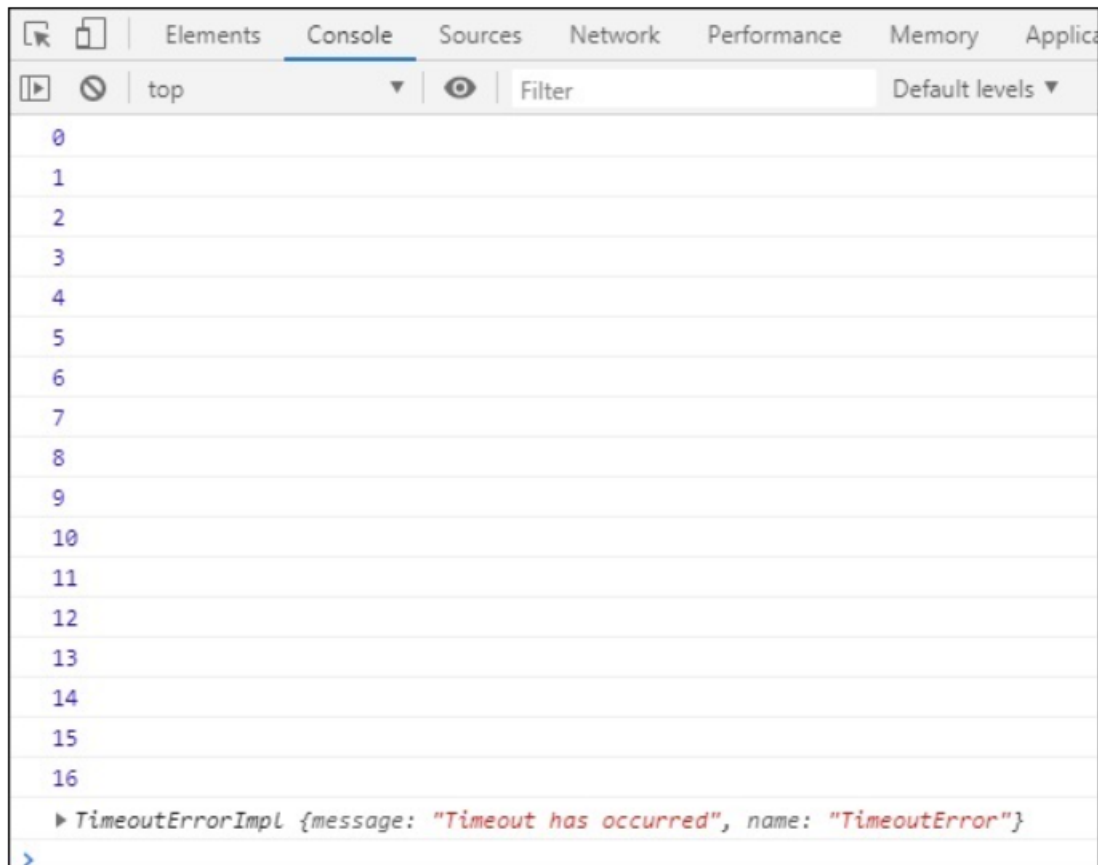
- időtúllépés - A bemenet az az időkorlát, amely típusszámú vagy dátumú lehet, amelyen belül az observable forrás értékét ki kell adni.

Observable eredményt adunk vissza, amely a megadott időkorlát alapján megáll.

```
import { of, interval } from 'rxjs';
import { filter, timeout } from 'rxjs/operators';

let list1 = interval(1000);
let final_val = list1.pipe(timeout(new Date("October 01, 2019 10:40:00")));
final_val.subscribe(
  x => console.log(x),
  e => console.log(e),
  () => console.log("Task complete")
);
```

Az observable intervallum folytatódik, és az időkorlátot új dátumként adják meg ("2019. október 01. 10:40:00"), tehát abban az időben bekövetkezik az időkorlát, és hibát dob az alábbiak szerint.



RxJS - Utility Operator toArray

`toArray():Observable`

Observable eredményt ad, amely a tömbként observable forrásból adja ki az értékeket, amikor a forrás elkészül.

```
import { of } from 'rxjs';
import { toArray } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let final_val = list1.pipe(toArray());
final_val.subscribe(x => console.log(x));
```

[2, 3, 4, 5, 6]

Conditional Operators

Sr.No	Operator & Description
1	<p>defaultIfEmpty</p> <p>Ez az operátor visszaad egy alapértelmezett értéket, ha az observable forrás üres.</p>
2	<p>every</p> <p>A bemeneti függvény alapján egy observable-t ad vissza, amely kielégíti az observable forrás minden egyes értékének feltételét.</p>
3	<p>find</p> <p>Ez visszaadja az observable-t, amikor az observable forrás első értéke megfelel a bemenetként vett predikátumfüggvény feltételének.</p>
4	<p>findIndex</p> <p>Ez a bemeneti ütemezőn alapuló operátor újból elküldi az Observable forrás értesítéseit.</p>
5	<p>isEmpty</p> <p>Ez az operátor akkor adja meg a kimenetet igaznak, ha az observable bemenet a complete callback híváshoz vezet, anélkül, hogy értékeket adna ki, és hamis, ha az observable bemenet bármilyen értéket bocsát ki.</p>

RxJS - Conditional Operator defaultIfEmpty

```
defaultIfEmpty(defaultValue = null): Observable
```

- **defaultValue** - Az **defaultValue** argumentum adhat némi értéket, vagy ha nincs megadva, akkor alapértelmezés szerint null.

Observable értéket ad vissza alapértelmezett értékkel, ha az observable forrás üres.

```
import { of } from 'rxjs';
import { defaultIfEmpty } from 'rxjs/operators';

let list1 = of();
let final_val = list1.pipe(defaultIfEmpty('Empty! No values'));
final_val.subscribe(x => console.log(x));
```

```
Empty! No values
```

RxJS - Conditional Operator every

```
every(predicate_func: function): Observable
```

- predicate_func - Az operátornak megadott input egy predicate_func, amely beveszi a forrás elemet, és ellenőrzi, hogy teljesíti-e a megadott feltételt.

A bemeneti függvény alapján egy observable-t ad vissza, amely kielégíti az observable forrás minden egyes értékének feltételét.

```
import { of } from 'rxjs';
import { every } from 'rxjs/operators';

let list1 = of(1, 3, 4, 9, 10, 15);
let final_val = list1.pipe(every(x => x % 2 === 0),);
final_val.subscribe(x => console.log(x));
```

false

RxJS - Conditional Operator find

```
find(predicate_func: function): Observable
```

- predicate_func - Az operátornak megadott input egy predicate_func, amely beveszi a forrás elemet, és ellenőrzi, hogy teljesíti-e a megadott feltételt.

Visszaadja az observable-t, amikor az observable forrás első értéke megfelel a bemenetként vett predikátumfüggvény feltételének.

```
import { of } from 'rxjs';
import { find } from 'rxjs/operators';

let list1 = of(24, 3, 4, 9, 10, 15);
let final_val = list1.pipe(find(x => x % 2 === 0),);
final_val.subscribe(x => console.log(x));
```

24

RxJS - Conditional Operator findIndex

```
findIndex(predicate_func: function): Observable
```

- predicate_func A predicate_function dönt az első indexről, amelyet a feltétel teljesül.

Observable értéket ad vissza az observable forrás első értékével, amely véletlenül kielégíti a predikátumfüggvény feltételét

```
import { of } from 'rxjs';
```

```
import { findIndex } from 'rxjs/operators';

let list1 = of(24, 3, 4, 9, 10, 15);
let final_val = list1.pipe(findIndex(x => x % 2 === 0),);
final_val.subscribe(x => console.log(x));
```

0

RxJS - Conditional Operator isEmpty

isEmpty(): Observable

Observable értéket ad vissza logikai értékkel igazként, ha az observable forrás üres, különben hamis.

```
import { of } from 'rxjs';
import { isEmpty } from 'rxjs/operators';

let list1 = of();
let final_val = list1.pipe(isEmpty(),);
final_val.subscribe(x => console.log(x));
```

true

Multicasting Operators

Sr.No	Operator & Description
1	<p>multicast</p> <p>A multicast megosztja a létrehozott egyetlen előfizetését más előfizetőkkel. A csoportos küldésű paraméterek olyan tsubject vagy factory metódusok, amelyek visszaadják a ConnectableObservable, amelynek connect () metódusát. A feliratkozáshoz a connect () metódust kell meghívni.</p>
2	<p>publish</p> <p>Ez az operátor visszaküldi a ConnectableObservable-t, és a connect () metódust kell használnia az observable feliratkozáshoz.</p>
3	<p>publishBehavior</p> <p>A publishBehaviour használja a BehaviourSubject alkalmazást, és visszaadja a ConnectableObservable parancsot. A connect () metódust kell használni, hogy feliratkozhasson az observable létrehozásra.</p>
4	<p>publishLast</p> <p>A publishBehaviour használja az AsyncSubject alkalmazást, és visszatér a</p>

	ConnectableObservable névre. A connect () metódust kell használni, hogy feliratkozhasson az observable létrehozottra.
5	publishReplay A publishReplay olyan viselkedéstárgyat használ, amelyben pufferolhatja az értékeket, és ugyanezt visszajátszhatja az új előfizetőknek, és visszatér a ConnectableObservable értékhez. A connect () metódust kell használni, hogy feliratkozhasson az observable létrehozottra.
6	share Ez egy alias a multicast () operátor számára, az egyetlen különbség az, hogy az előfizetés megkezdéséhez nem kell manuálisan meghívunk a connect () metódust.

RxJS - Multicasting Operator multicast

```
multicast(subjectOrSubjectFactory: Subject): OperatorFunction
```

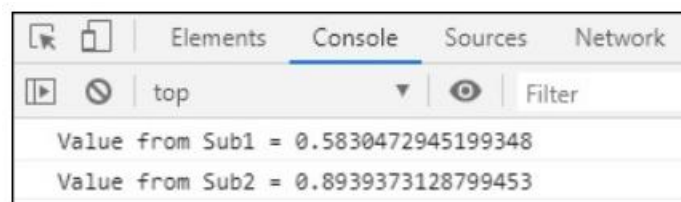
- subjectOrSubjectFactory: a multicast-nak átadott paraméter egy subject vagy factory metódus, amely egy subject-et ad vissza.

Mielőtt belekezdnenénk egy multicast () operátor működésébe, először értsük meg, hogy a multicast () operátor hasznos.

```
import { Observable } from 'rxjs';

var observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(Math.random());
  } catch (e) {
    subscriber.error(e);
  }
});

const subscribe_one = observable.subscribe(val => console.log(
  "Value from Sub1 = "+val)
);
const subscribe_two = observable.subscribe(val => console.log(
  "Value from Sub2 = "+val)
);
```



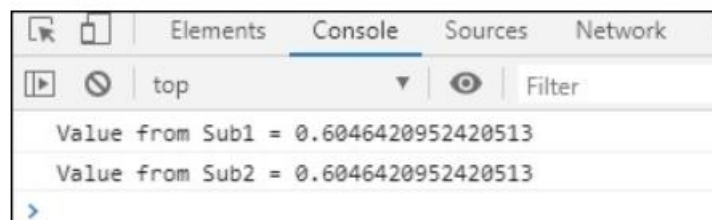
Ha látja a kimenetet, az Sub1 és Sub2 értékei eltérnek. Ez azért van, mert amikor a subscriber-t hívják, az observable újraindítás megadja a rendelkezésre álló friss értéket. De szükségünk van arra, hogy a hívott subscriber-ek azonos értékűek legyenek.

Itt van egy multicast () operátor, amely segítségünkre van.

```
import { Observable, Subject } from 'rxjs';
import { take, multicast, mapTo } from 'rxjs/operators';

var observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(Math.random());
  } catch (e) {
    subscriber.error(e);
  }
});
const multi_op = observable.pipe(multicast(() => new Subject()));
const subscribe_one = multi_op.subscribe(
  x => console.log("Value from Sub1 = "+x)
);
const subscribe_two = multi_op.subscribe(
  x => console.log("Value from Sub2 = "+x)
);
multi_op.connect();
```

Ha most látja, hogy ugyanaz az érték van megosztva a hívott subscriber-ek között.



RxJS - Multicasting Operator publish

publish()

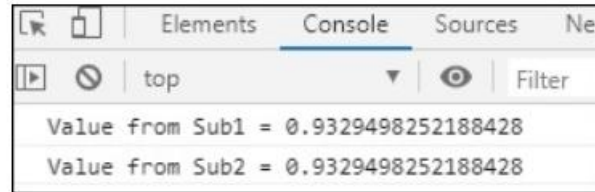
```
import { interval, Observable } from 'rxjs';
import { filter, publish } from 'rxjs/operators';
var observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(Math.random());
  } catch (e) {
    subscriber.error(e);
  }
});
const observable1 = publish()(observable);
const subscribe_one = observable1.subscribe(
  x => console.log("Value from Sub1 = "+x)
```



```

);
const subscribe_two = observable1.subscribe(
  x => console.log("Value from Sub2 = "+x)
);
observable1.connect();

```



RxJS - Multicasting Operator publishBehavior

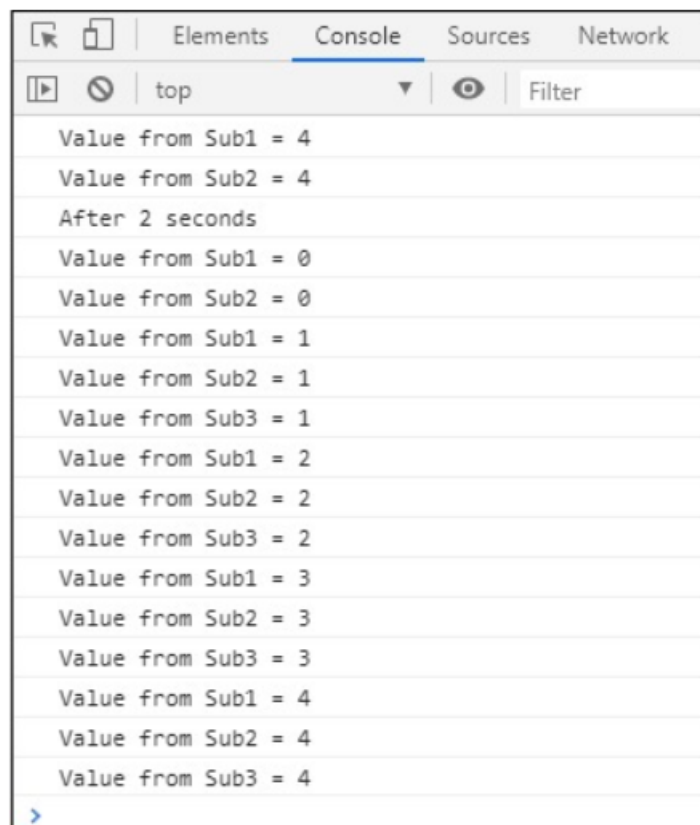
publishBehaviour(defaultvalue)

```

import { interval } from 'rxjs';
import { take, publishBehavior } from 'rxjs/operators';

let observer = interval(1000).pipe(
  take(5),
  publishBehavior(4)
);
const subscribe_one = observer.subscribe(
  x => console.log("Value from Sub1 = "+x)
);
const subscribe_two = observer.subscribe(
  x => console.log("Value from Sub2 = "+x)
);
observer.connect();
console.log("After 2 seconds");
setTimeout(() => {
  const subscribe_three = observer.subscribe(
    x => console.log("Value from Sub3 = "+x)
  );
}, 2000);

```



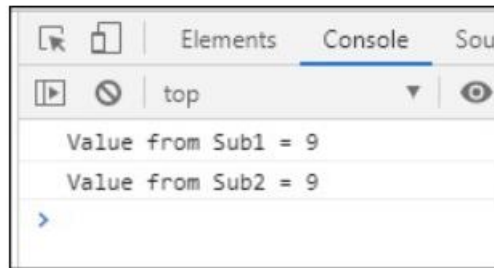
Az alapértelmezett érték jelenik meg először, majd később az observable érték.

RxJS - Multicasting Operator publishLast

A `publishBehaviour` használja az `AsyncSubject` alkalmazást, és visszaadja a `ConnectableObservable` parancsot. A `connect()` metódust kell használni, hogy feliratkozhasson az observable létrehozásra.

```
import { interval } from 'rxjs';
import { take, publishLast } from 'rxjs/operators';

let observer = interval(1000).pipe(
  take(10),
  publishLast()
);
const subscribe_one = observer.subscribe(
  x => console.log("Value from Sub1 = "+x)
);
const subscribe_two = observer.subscribe(
  x => console.log("Value from Sub2 = "+x)
);
observer.connect();
```

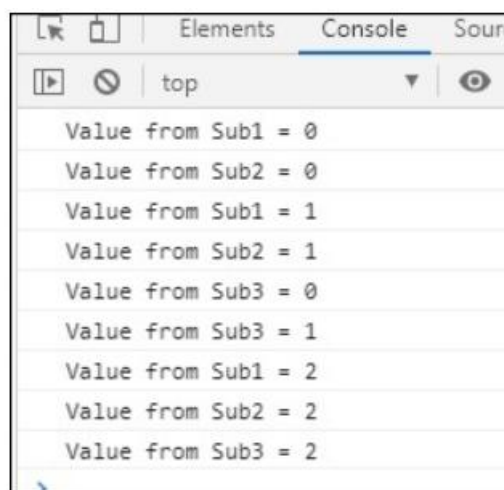


RxJS - Multicasting Operator publishReplay

`publishReplay(value);` // here value is the number of times it has to replay.

```
import { interval } from 'rxjs';
import { take, publishReplay } from 'rxjs/operators';

let observer = interval(1000).pipe(
  take(3),
  publishReplay(2)
);
const subscribe_one = observer.subscribe(
  x => console.log("Value from Sub1 = "+x)
);
const subscribe_two = observer.subscribe(
  x => console.log("Value from Sub2 = "+x)
);
observer.connect();
setTimeout(() => {
  const subscribe_three = observer.subscribe(
    x => console.log("Value from Sub3 = "+x)
  );
}, 2000);
```



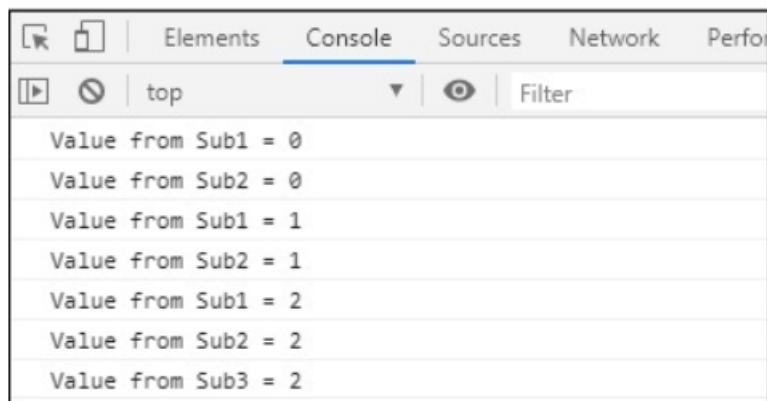
RxJS - Multicasting Operator share

Ez egy alias a multicast () operátor számára, az egyetlen különbség az, hogy az előfizetés megkezdéséhez nem kell manuálisan meghívunk a connect () metódust.

share()

```
import { interval } from 'rxjs';
import { take, share } from 'rxjs/operators';

let observer = interval(1000).pipe(take(3), share());
const subscribe_one = observer.subscribe(
  x => console.log("Value from Sub1 = "+x)
);
const subscribe_two = observer.subscribe(
  x => console.log("Value from Sub2 = "+x)
);
setTimeout(() => {
  const subscribe_three = observer.subscribe(
    x => console.log("Value from Sub3 = "+x)
  );
}, 2000);
```



Error Handling Operators

Sr.No	Operator & Description
1	<p>catchError</p> <p>Ez az operátor gondoskodik az Observable forrás hibáinak elkapásáról egy új Observable vagy hiba visszaadásával.</p>
2	<p>retry</p> <p>Ez az operátor gondoskodik az observable forrás újrapróbálásáról, ha hiba van, és az újrapróbálkozás a megadott bemeneti szám alapján történik.</p>

RxJS - Error Handling Operator catchError

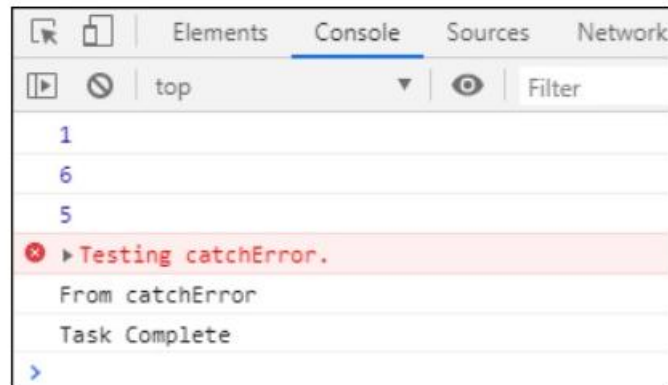
```
catchError(selector_func: (err_func: any, caught: Observable) => Observable)
```

- **selector_func** - A szelektor func 2 argumentumot, hibafüggvényt és befogott értéket vesz fel, ami observable.

Observable-t ad vissza a selector_func által kibocsátott érték alapján.

```
import { of } from 'rxjs';
import { map, filter, catchError } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(
  map(el => {
    if (el === 10) {
      throw new Error("Testing catchError.");
    }
    return el;
  }),
  catchError(err => {
    console.error(err.message);
    return of("From catchError");
  })
);
final_val.subscribe(
  x => console.log(x),
  err => console.error(err),
  () => console.log("Task Complete")
);
```



RxJS - Error Handling Operator retry

```
retry(retry_count: number): Observable
```

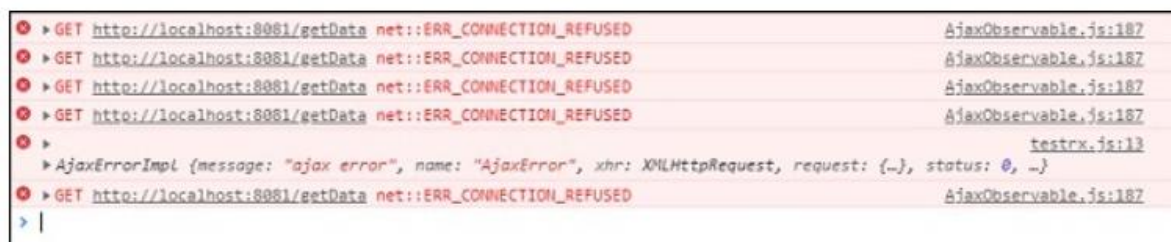
- **retry_count** - A retry_count argumentum az újrapróbálkozások száma.

Visszaadja az újrapróbálkozási logikával observable forrást.

```
import { of } from 'rxjs';
import { map, retry } from 'rxjs/operators';
import { ajax } from 'rxjs/ajax';

let all_nums = of(1, 6, 5, 10, 9, 20, 10);
let final_val = ajax('http://localhost:8081/getData').pipe(retry(4));
final_val.subscribe(
  x => console.log(x), => console.error(err),
  () => console.log("Task Complete")
);
```

A példában az ajax használatával hívunk egy URL-t. Az URL - http://localhost:8081/getData 404-et ad, ezért a retry () operátor négyszer megpróbálja újra meghívni az URL-t. A kimenet az alábbiakban látható



```
> GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
> GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
> GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
> GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
> AjaxErrorImpl {message: "ajax error", name: "AjaxError", xhr: XMLHttpRequest, request: {_, status: 0, _}} testrx.js:13
> GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
> |
```

RxJS - Working with Subscription

Amikor az observable létrejön, az observable végrehajtásához fel kell iratkoznunk (subscribe) rá.

count() operator

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';

let all_nums = of(1, 7, 5, 10, 10, 20);
let final_val = all_nums.pipe(count());
final_val.subscribe(x => console.log("The count is "+x));
```

The count is 6

Az előfizetésnek (subscription) van egy metódusa, amelyet unsubscribe () néven hívnak. A unsubscribe () módszerrel történő hívás eltávolítja az összes observable elemhez felhasznált erőforrást, vagyis az observable törlődik. Itt egy működő példa az unsubscribe () módszer használatára.

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';

let all_nums = of(1, 7, 5, 10, 10, 20);
let final_val = all_nums.pipe(count());
let test = final_val.subscribe(x => console.log("The count is "+x));
```

```
test.unsubscribe();
```

Az előfizetés (subscription) a változó tesztben tárolódik. Használtuk a test.unsubscribe ()-t is.

The count is 6

RxJS - Working with Subjects

A subject observable, amely képes multicast küldésre, azaz sok observable-el beszélgetni. Vegyünk egy gombot egy eseményfigyelővel (event listener), az eseményhez csatolt függvényt hozzáadjuk a figyelő (listener) hozzáadásához, minden alkalommal, amikor a felhasználó rákattint a gombra, hasonló funkciók vonatkoznak a témára is.

Create a subject

A subject használatához be kell importálnunk

```
import { Subject } from 'rxjs';
```

Subject objektumot a következőképpen hozhat létre:

```
const subject_test = new Subject();
```

Az objektum egy observable, amelynek három metódusa van –

- next(v)
- error(e)
- complete()

Subscribe to a Subject

Több feliratkozást (subscription) is létrehozhat a témában (subject) az alábbiak szerint –

```
subject_test.subscribe({
  next: (v) => console.log(`From Subject : ${v}`)
});
subject_test.subscribe({
  next: (v) => console.log(`From Subject: ${v}`)
});
```

Az előfizetés (subscription) regisztrálva van a tárgy (subject) objektumra, csakúgy, mint a korábban tárgyalt addlistener.

Passing Data to Subject

Adatokat továbbíthat a `next ()` módszerrel létrehozott subject-nek.

```
subject_test.next("A");
```

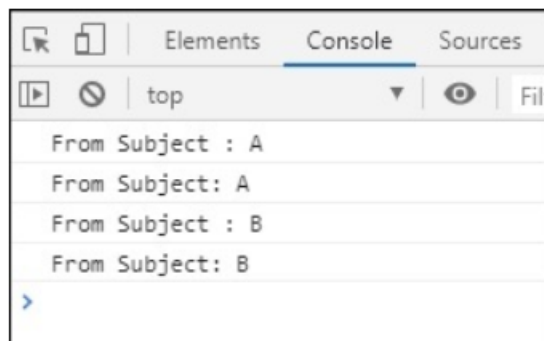
Az adatokat továbbítjuk a tárgyhoz (subject) hozzáadott összes előfizetéshez (subscription).

```
import { Subject } from 'rxjs';

const subject_test = new Subject();

subject_test.subscribe({
  next: (v) => console.log(`From Subject : ${v}`)
});
subject_test.subscribe({
  next: (v) => console.log(`From Subject: ${v}`)
});
subject_test.next("A");
subject_test.next("B");
```

Az `subject_test` objektum egy `new Subject ()` meghívásával jön létre. Az `subject_test` objektumnak van referenciája a `next ()`, `error ()` és a `complete ()` metódusokra.



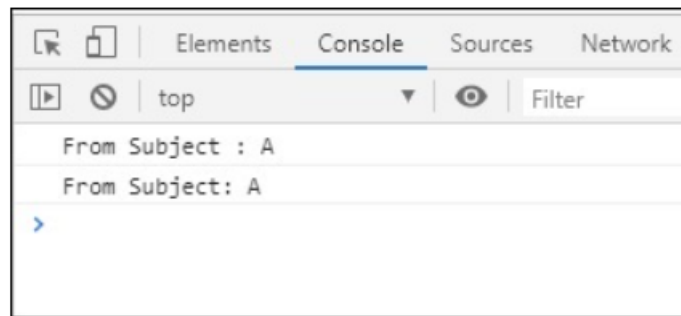
Használhatjuk a `complete ()` metódust a subject végrehajtásának leállításához az alábbiak szerint.

```
import { Subject } from 'rxjs';

const subject_test = new Subject();

subject_test.subscribe({
  next: (v) => console.log(`From Subject : ${v}`)
});
subject_test.subscribe({
  next: (v) => console.log(`From Subject: ${v}`)
});
subject_test.next("A");
subject_test.complete();
subject_test.next("B");
```

Amint a `complete`-et hívjuk a következő később meghívott módszert nem hívjuk meg.

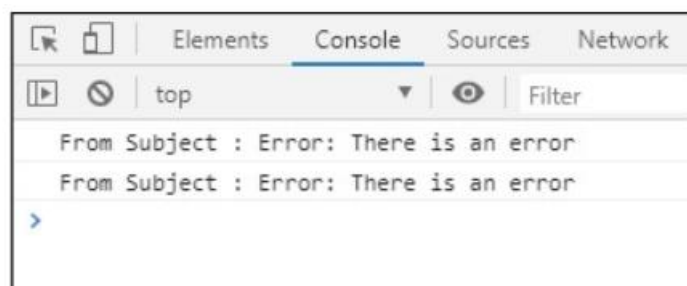


Nézzük meg, hogyan hívjuk meg a `error()` metódust.

```
import { Subject } from 'rxjs';

const subject_test = new Subject();

subject_test.subscribe({
  error: (e) => console.log(`From Subject : ${e}`)
});
subject_test.subscribe({
  error: (e) => console.log(`From Subject : ${e}`)
});
subject_test.error(new Error("There is an error"));
```



What is the Difference between Observable and Subject?

Egy observable beszélgetni fog egy-egy subscriber-rel. Bármikor feliratkozik az observable-re, a végrehajtás a semmiből indul. Hívjon egy Http-hívást, amelyet ajax használatával indítottak, és 2 előfizetőt (subscriber) hívtak az observable felé. 2 Http kérést fog látni a böngésző hálózati lapján.

```
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators';

let final_val = ajax('https://jsonplaceholder.typicode.com/users').pipe(map(e => e.response));
let subscriber1 = final_val.subscribe(a => console.log(a));
let subscriber2 = final_val.subscribe(a => console.log(a));
```

Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	200	document	Other	729 B	337 ms	
main_bundle.js	200	script	(index)	570 KB	15 ms	
users	200	xhr	AjaxObservable.js:187	2.2 KB	550 ms	
users	200	xhr	AjaxObservable.js:187	44 B	715 ms	

```

▶ (10) [{-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}]
▶ (10) [{-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}, {-}]
>

```

Most itt van a probléma, hogy ugyanazokat az adatokat szeretnénk megosztani, de nem 2 Http hívás árán. Szeretnénk egy Http-hívást kezdeményezni és megosztani az adatokat az előfizetők (subscriber) között.

Ez a subject-ek használatával lehetséges. Observable, amely képes multicast küldésre, azaz sok observable-lel beszélgetni. Megoszthatja az értéket az előfizetők (subscriber) között.

```

import { Subject } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators';

const subject_test = new Subject();

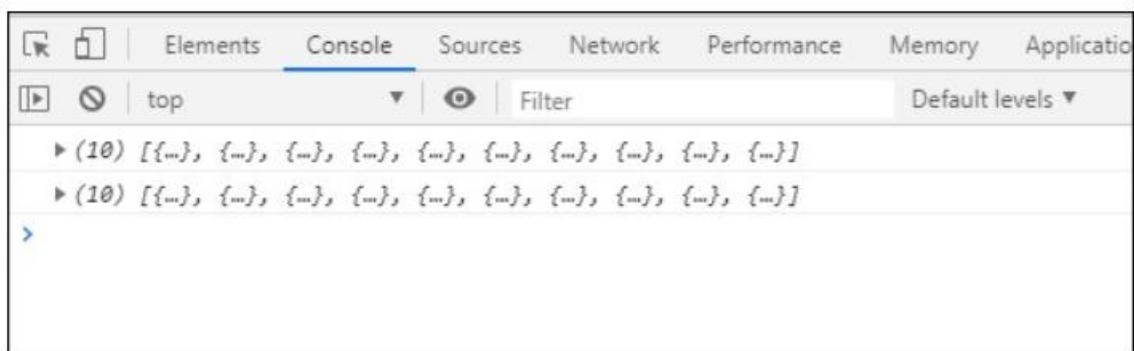
subject_test.subscribe({
  next: (v) => console.log(v)
});
subject_test.subscribe({
  next: (v) => console.log(v)
});

let final_val = ajax('https://jsonplaceholder.typicode.com/users').pipe(map(e => e.response));
let subscriber = final_val.subscribe(subject_test);

```

Name	Status	Type	Initiator	Size	Time
localhost	304	document	Other	242 B	
main_bundle.js	200	script	(index)	650 KB	
users	200	xhr	AjaxObservable.js:187	462 B	

Most csak egy Http-hívást láthat, és ugyanazok az adatok vannak megosztva a hívott előfizetők (subscriber) között.



Behaviour Subject

A behaviour subject a legfrissebb értéket adja meg, amikor hívják.

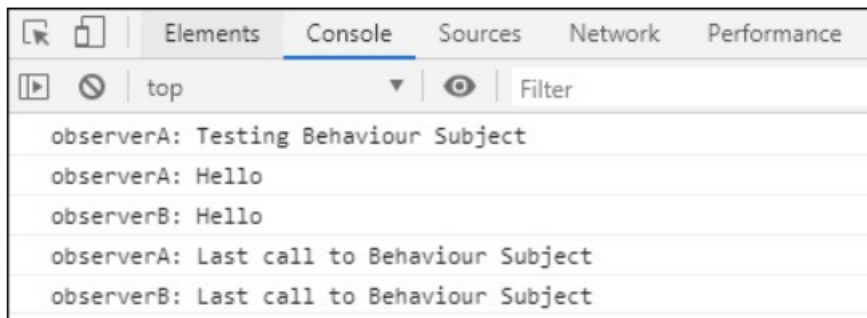
Létrehozhat behaviour subject-et az alábbiak szerint –

```
import { BehaviorSubject } from 'rxjs';
const subject = new BehaviorSubject("Testing Behaviour Subject");
// initialized the behaviour subject with value:Testing Behaviour Subject
```

```
import { BehaviorSubject } from 'rxjs';
const behavior_subject = new BehaviorSubject("Testing Behaviour Subject");
// 0 is the initial value

behavior_subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

behavior_subject.next("Hello");
behavior_subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});
behavior_subject.next("Last call to Behaviour Subject");
```



Replay Subject

Egy replay subject hasonló a behaviour subject-hez, ahol pufferolhatja az értékeket, és ugyanezt visszajátszhatja az új előfizetőknek.

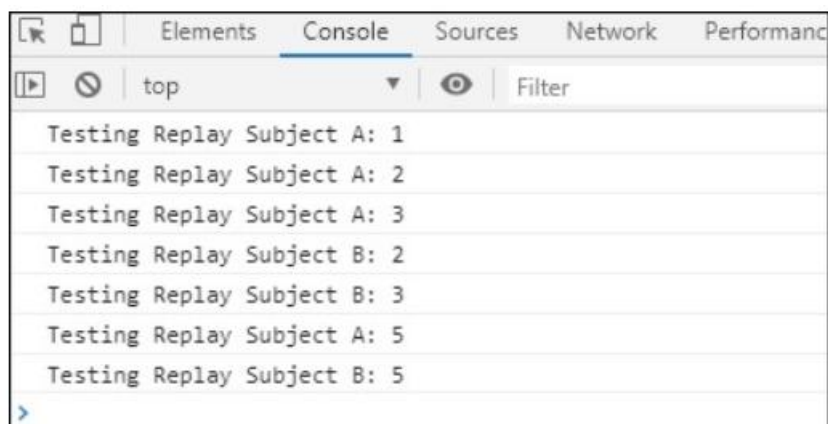
```
import { ReplaySubject } from 'rxjs';
const replay_subject = new ReplaySubject(2);
// buffer 2 values but new subscribers

replay_subject.subscribe({
  next: (v) => console.log(`Testing Replay Subject A: ${v}`)
});

replay_subject.next(1);
replay_subject.next(2);
replay_subject.next(3);
replay_subject.subscribe({
  next: (v) => console.log(`Testing Replay Subject B: ${v}`)
});

replay_subject.next(5);
```

Az alkalmazott pufferérték 2 az replay subject-en. Tehát az utolsó két értéket pufferolja és felhasználják az új hívott előfizetők számára.



AsyncSubject

Az AsyncSubject esetében az utoljára meghívott értéket továbbítják az előfizetőnek, és ez csak a complete() metódus meghívása után történik meg.

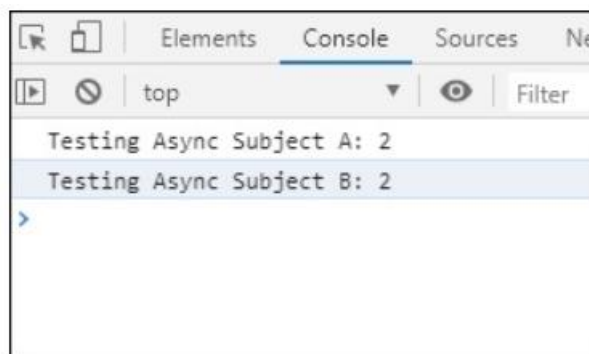
```
import { AsyncSubject } from 'rxjs';

const async_subject = new AsyncSubject();

async_subject.subscribe({
  next: (v) => console.log(`Testing Async Subject A: ${v}`)
});

async_subject.next(1);
async_subject.next(2);
async_subject.complete();
async_subject.subscribe({
  next: (v) => console.log(`Testing Async Subject B: ${v}`)
});
```

Itt, mielőtt a teljeset nevezzük, a subject-nek utoljára átadott érték 2 és ugyanaz, amelyet az előfizetőknek (subscriber) adott.



RxJS - Working with Scheduler

Az ütemező (scheduler) ellenőrzi az előfizetés (subscription) kezdetének és értesítésének végrehajtását.

Az ütemező használatához a következőkre van szükségünk:

```
import { Observable, asyncScheduler } from 'rxjs';
import { observeOn } from 'rxjs/operators';
```

```
import { Observable, asyncScheduler } from 'rxjs';
import { observeOn } from 'rxjs/operators';

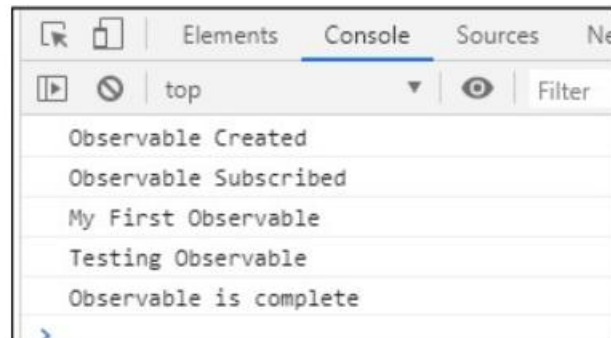
var observable = new Observable(function subscribe(subscriber) {
  subscriber.next("My First Observable");
  subscriber.next("Testing Observable");
});
```

```

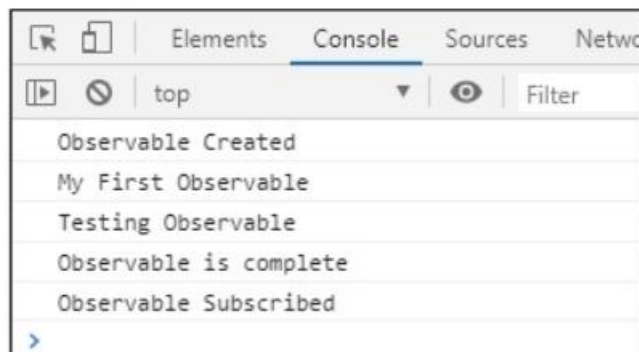
    subscriber.complete();
  }).pipe(
    observeOn(asyncScheduler)
  );
console.log("Observable Created");
observable.subscribe(
  x => console.log(x),
  (e)=>console.log(e),
  ()=>console.log("Observable is complete")
);

console.log('Observable Subscribed');

```



Scheduler nélkül pedig így nézne ki:



Working with RxJS & Angular

app.component.ts

```

import { Component } from '@angular/core';
import { environment } from '../environments/environment';
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators'

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = '';
  data;
  constructor() {

```

```

    this.data = "";
    this.title = "Using RxJs with Angular";
    let a = this.getData();
  }
  getData() {
    const response =
      ajax('https://jsonplaceholder.typicode.com/users')
        .pipe(map(e => e.response));
    response.subscribe(res => {
      console.log(res);
      this.data = res;
    });
  }
}

```

app.component.html

```

<div>
  <h3>{{title}}</h3>
  <ul *ngFor="let i of data">
    <li>{{i.id}}: {{i.name}}</li>
  </ul>
</div>

<router-outlet></router-outlet>

```

