

Observables and RxJS

Лектор: Петър Маламов

Reactive Programming

Какво е Reactive Programming ?

Парадигма за програмиране, която се фокусира върху асинхронното обработване на потоци от данни, където промени в данните автоматично предизвикват промени в зависимите компоненти.

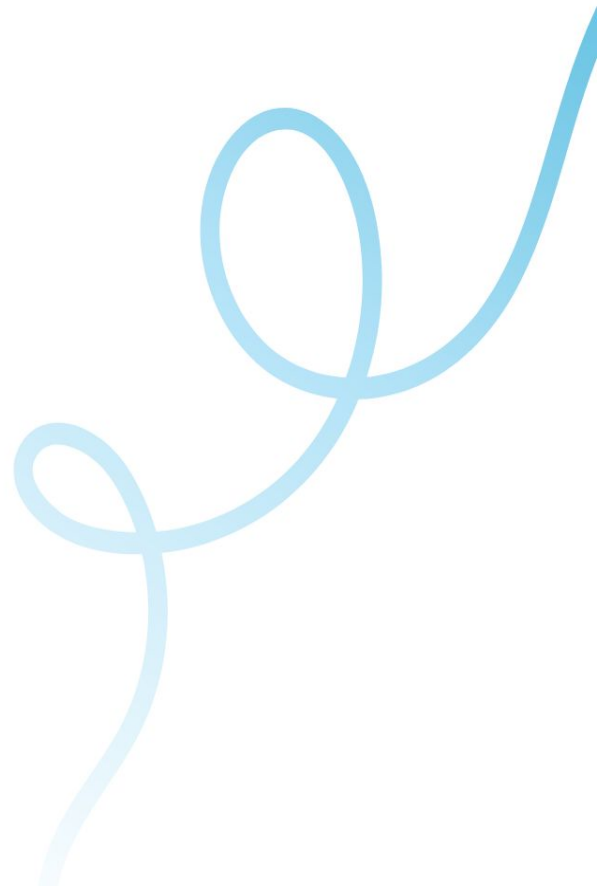
Основната идея е реакция на събития, системата автоматично реагира на промени, като ги разпространява през съответните компоненти.

Ползи

- Намалява свързаността между класовете (Loose Coupling)
- Повишава гъвкавостта
- Улеснява поддръжка

Негативи

- Learning Curve
- ПОВИШЕНА СЛОЖНОСТ



Кога да използваме Reactive Programming ?

- Когато промените в състоянието на един обект може да изискват промяна на други обекти
- Когато някои обекти в приложението трябва да наблюдават други, но само за ограничен период от време или в специфични случаи

RxJS

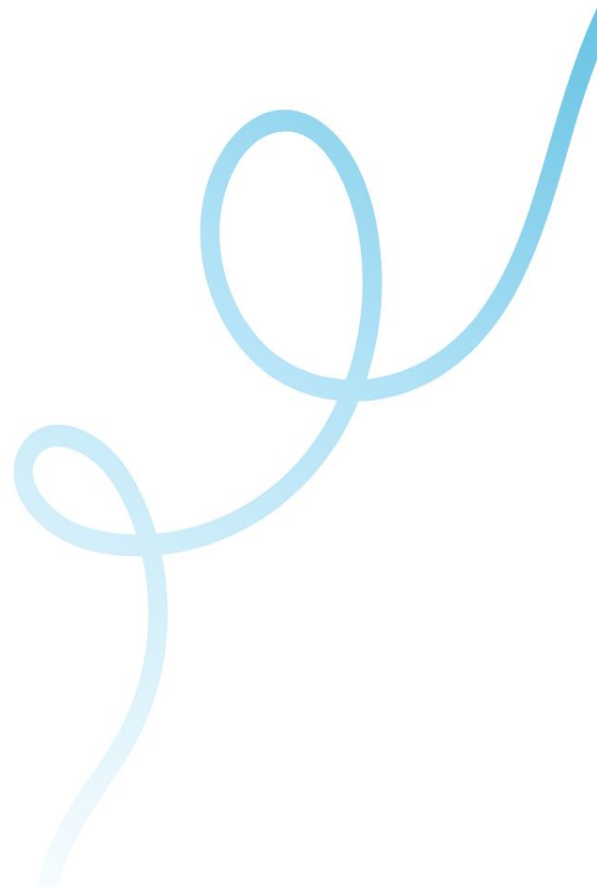
Какво е RxJS ?

Библиотека за съставяне на асинхронни или event-based програми, чрез използване на “проследими” потоци от данни.

Тя позволява работа с асинхронни данни и събития чрез използване на функционални техники

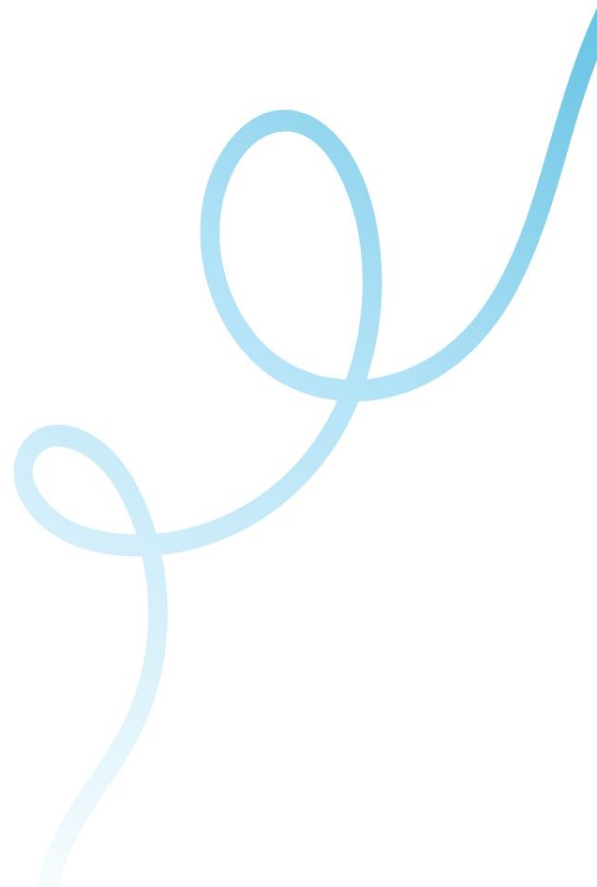
Как се ползва от Angular ?

- Routing events
- Reactive form events
- Http responses



Основни концепции в RxJS

- Observable
- Observer
- Subscription
- Operators
- Subject



Observable

Източник на асинхронни или синхронни потоци от данни. Тези потоци мога да бъдат безкрайни или ограничени.

Може да изпраща три типа известия:

- `next` - Уведомява за нов елемент в потока.
- `error` - Уведомява за грешка.
- `complete` - Уведомява за приключване на потока.

Данните в един Observable мога да бъдат:

- Примитиви - числа, низове и др.
- Events - mouse, key, valueChanges, routing
- Обекти, масиви или други observables
- HTTP response

TS app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { from, fromEvent, of } from 'rxjs';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  title = '05-observables-and-rxjs';

  ngOnInit(): void {
    // краен брой елементи
    of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
    //

    // безкраен брой елементи
    fromEvent(document, 'click');
    //
  }
}
```

Observer

Обект, който наблюдава и реагира на известия от потоци от данни.

Те имат три основни метода:

- `next` — за обработка на ново получена стойност.
- `error` — за обработка на грешки.
- `complete` — когато потокът е завършен.

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  title = '05-observables-and-rxjs';

  ngOnInit(): void {
    // създаване на observer
    const observer: Observer<any> = {
      next: (value) => console.log(value),
      error: (err) => console.log(err),
      complete: () => console.log('complete'),
    };

    // краен брой елементи
    of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    //

    // безкраен брой елементи
    fromEvent(document, 'click');
    //
  }
}
```

Subscription

Представява връзката между **Observable** и **Observer**.

Когато **Observer** се "абонира" за **Observable**, той започва да получава емитираните стойности, грешки или сигнали за завършване.

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  title = '05-observables-and-rxjs';

  ngOnInit(): void {
    // създаване на observer
    const observer: Observer<any> = {
      next: (value) => console.log(value),
      error: (err) => console.log(err),
      complete: () => console.log('complete'),
    };

    // краен брой елементи
    of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).subscribe(observer);

    from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).subscribe(observer);
    //

    // безкраен брой елементи
    fromEvent(document, 'click').subscribe(observer);
    //
  }
}
```

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  title = '05-observables-and-rxjs';

  ngOnInit(): void {
    // краен брой елементи
    of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).subscribe(
      (nextValue) => console.log(nextValue) // callback за вземане на следващата стойност
    );

    from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).subscribe(
      (nextValue) => console.log(nextValue)
    );
    //

    // безкраен брой елементи
    fromEvent(document, 'click').subscribe(
      (nextValue) => console.log(nextValue)
    );
    //
  }
}
```

Важно!

Винаги затваряйте абонаментите, за да предотвратите изтичане на памет.

Ако не се отписвате от дълготрайни или многократни потоци, те могат да продължат да съществуват и да консумират ресурси.

Когато създадем абонамент, получаваме функция за unsubscribe, която ни позволява да прекратим абонамента и да спрем получаването на емитирани стойности.

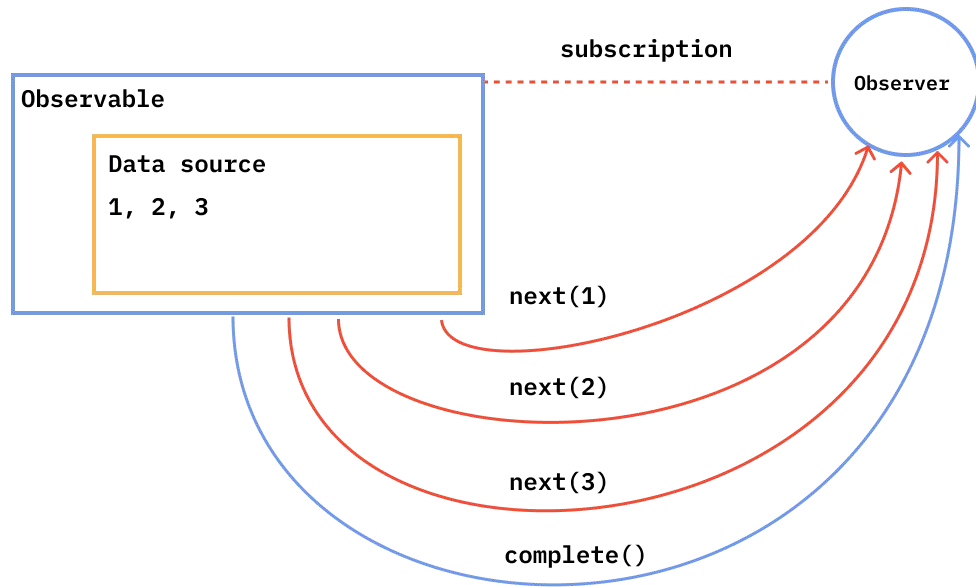
TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit, OnDestroy {
  title = '05-observables-and-rxjs';
  subscriptions: Subscription[] = [];

  ngOnInit(): void {
    const sub = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).subscribe((nextValue) => console.log(nextValue));

    this.subscriptions.push(sub);
  }

  ngOnDestroy(): void {
    this.subscriptions.forEach((sub) => sub.unsubscribe());
  }
}
```



Operators

Функции, които взима Observable като вход и връща нов Observable като изход, трансформирайки потока от данни.

Операторите позволяват да се променят, филтрират, комбинират или манипулират данните в потоците, преди да бъдат изпратени до наблюдателите.

Можем да прилагаме оператори последователно, използвайки метода **pipe()** на Observable.

TS app.component.ts

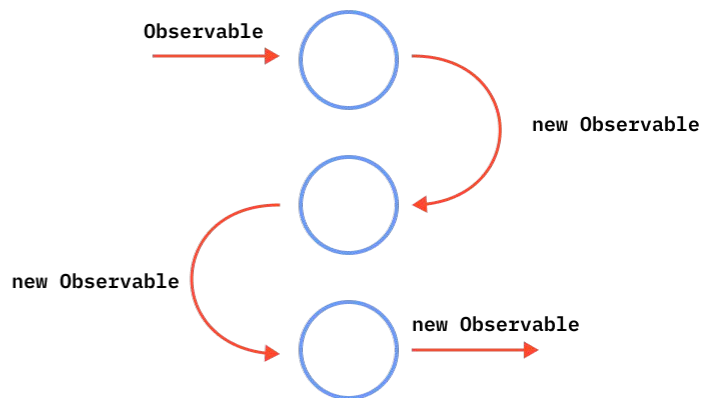
```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  title = '05-observables-and-rxjs';





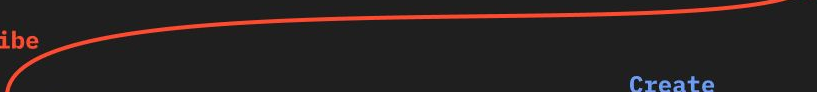


  ngOnInit(): void {
    of(1, 2, 3, 4)
      .pipe(
        filter((x) => x % 2 === 0),
        map((x) => x * x),
        tap((x) => console.log('x', x))
      )
      .subscribe((nextValue) => console.log(nextValue));
  }
}
```

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  title = '05-observables-and-rxjs';

  ngOnInit(): void {
    of(1, 2, 3, 4)
      .pipe(
        filter((x) => x % 2 === 0),
        map((x) => x * x),
        tap((x) => console.log('x', x))
      )
      .subscribe((nextValue) => console.log(nextValue));
  }
}
```




```
of(1, 2, 3, 4)
  .pipe(
    Observable
      
      filter((x) => x % 2 === 0),  Observable
    
    map((x) => x * x),  Observable
    
    tap((x) => console.log('x', x))  Observable
    
    .subscribe((nextValue) => console.log(nextValue));
```

Важно!

Трябва да се опитваме да минимизираме броя на оператори които използваме.

TS app.config.ts

```
of(1, 2, 3, 4)
  .pipe(
    map((x) => x * x),
    map((x) => 'Number of items: ' + x),
    tap((x) => console.log('x', x))
  )
  .subscribe((nextValue) => console.log(nextValue));
```

TS app.config.ts

```
of(1, 2, 3, 4)
  .pipe(
    map((x) => {
      const newValue = x * x;
      return 'Number of items:' + newValue;
    }),
    tap((x) => console.log('x', x))
  )
  .subscribe((nextValue) => console.log(nextValue));
```

Subject

Специален тип Observable, който позволява едновременно изпращане на стойности към много Observers.

Всеки Subject е както Observable, така и Observer.

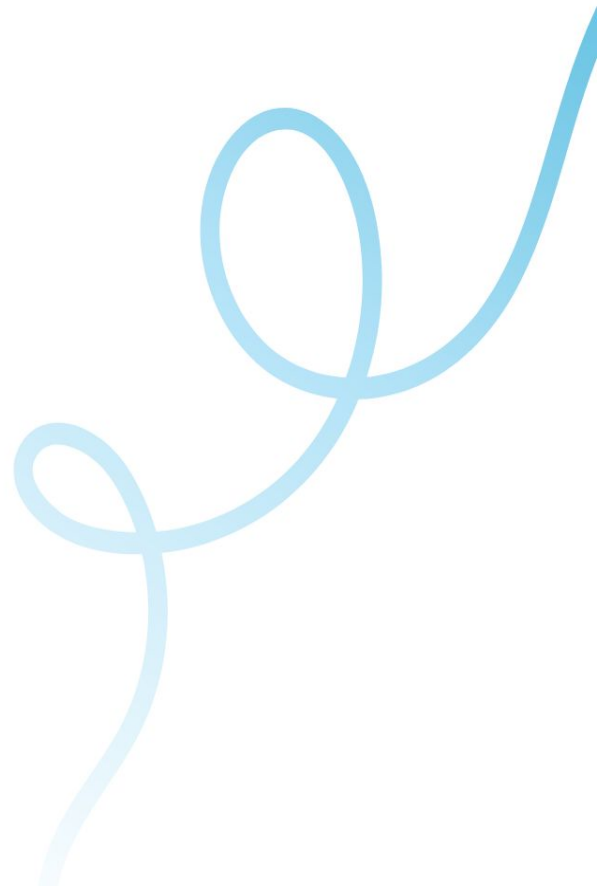
Можете да се абонирате за Subject, както и да извикате next, за да подавате стойности, както и error и complete, за да сигнализирате грешки или завършване на потока.

Типове Subjects

- **BehaviorSubject** - Съхранява последната емитирана стойност и я изпраща на новите абонати веднага след абониране.
- **ReplaySubject** - Съхранява определен брой емитирани стойности и ги изпраща на новите абонати.
- **AsyncSubject** - Емитира само последната стойност и то само след като потокът завърши (complete).

Типове Observables

- Cold Observables
- Hot Observables



Cold Observables

При студените observables, всеки абонат получава собствена, независима версия на потока от данни

.

При всеки нов абонамент се стартира нова екзекуция на потока.

В резултат на това стойностите няма да се изпращат едновременно на всички абонати. Всеки абонат ще получава стойностите спрямо момента, в който се е абонира.

TS app.component.ts

```
const obs1 = of(1, 2, 3);  
  
obs1.subscribe((value) => console.log(value));  
obs1.subscribe((value) => console.log(value));
```

Hot Observables

При горещите observables всички абонати споделят един и същ поток от данни.

Стойностите се предават към всички абонати едновременно, без значение кога са се абонирали.

Пример за горещ observable е поток от данни от събития в реално време, като кликване на бутон, уебсокет или стрийм от камерата.

TS app.component.ts

```
export class AppComponent implements OnInit {  
  title = '05-observables-and-rxjs';  
  subject = new Subject<number>();  
  obs$ = this.subject.asObservable();  
  constructor() {}  
  
  ngOnInit(): void {  
    this.obs$.subscribe((value) => console.log('a', value));  
    this.obs$.subscribe((value) => console.log('b', value));  
  
    this.subject.next(1);  
    this.subject.next(2);  
    this.subject.next(3);  
  }  
}
```

Console

a 1
b 1
a 2
b 2
a 3
b 3

RxJS Operators

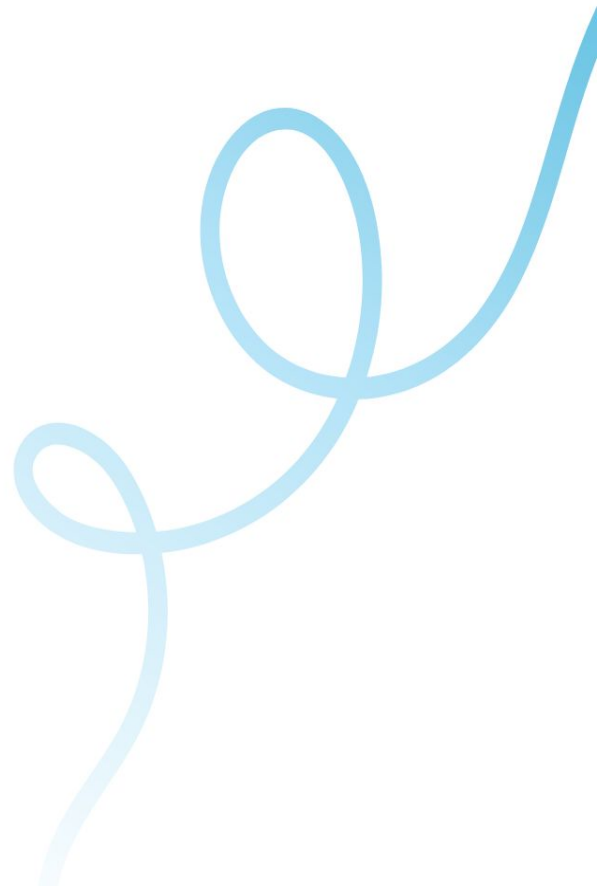
Типове RxJS Operators

- **Creation Operators** - from, fromEvent, interval, of и др.
- **Combination Operators** - combineLatest, concat, merge и др.
- **Transformation Operators** - map, concatMap, mergeMap и др.
- **Filtering Operators** - filter, take, first, last и др.
- **Join Operators** - concatAll, mergeAll, withLatestFrom и др.

- **Multicasting Operators** - multicast, share и др.
- **Error Handling Operators** - catchError, retry, retryWhen
- **Utility Operators** - tap, delay, toArray и др.
- **Conditional and Boolean Operators** - isEmpty, findIndex и др.
- **Mathematical and Aggregate Operators** - min, max, count и др.

Често използвани оператори

- map
- filter
- tap
- take

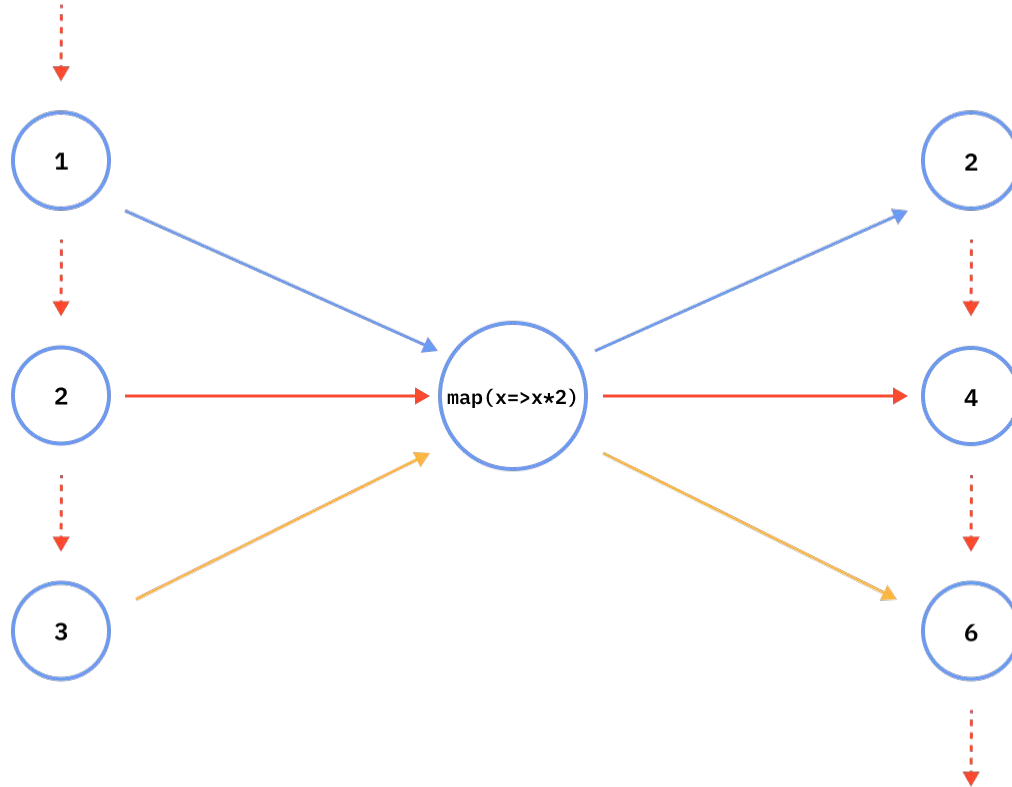


map

Използва се за трансформиране на елементите в потока.

При всяко получаване на стойност от потока, **map** прилага функция върху нея и връща резултата в потока.

Observable([1,2,3])



Observable([2,4,6])

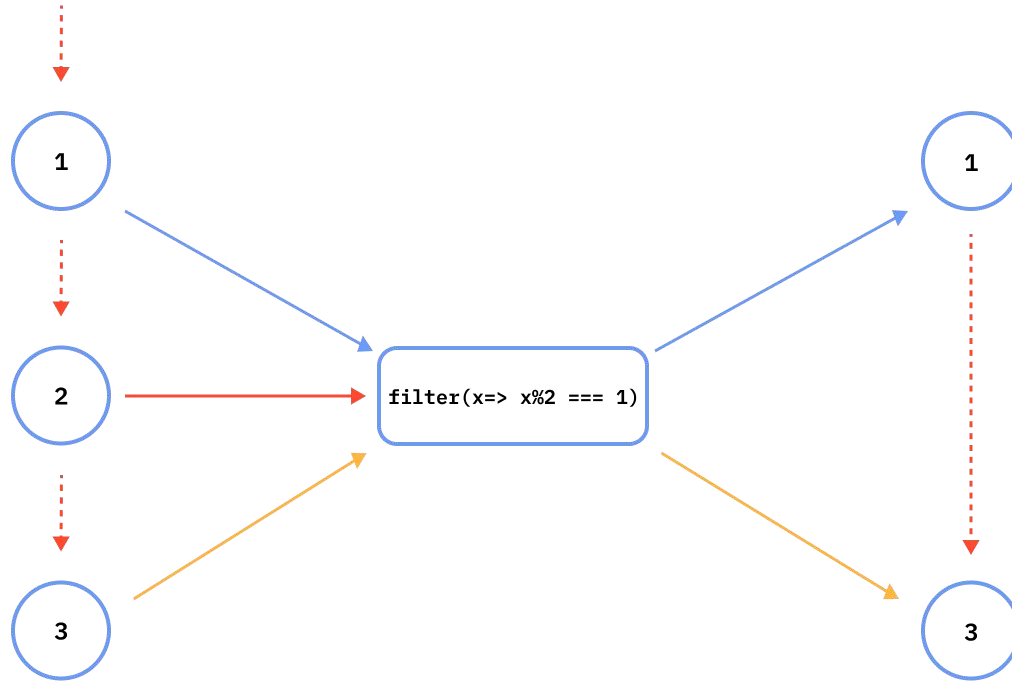
TS app.component.ts

```
of(1, 2, 3)  
  .pipe(map((x) => x * 2))  
  .subscribe((nextValue) => console.log(nextValue));
```


filter

Използва се за филтриране на елементите в потока, като оставя само тези, които отговарят на критериите от подадената функция.

Observable([1,2,3])



Observable([1,3])

TS app.component.ts

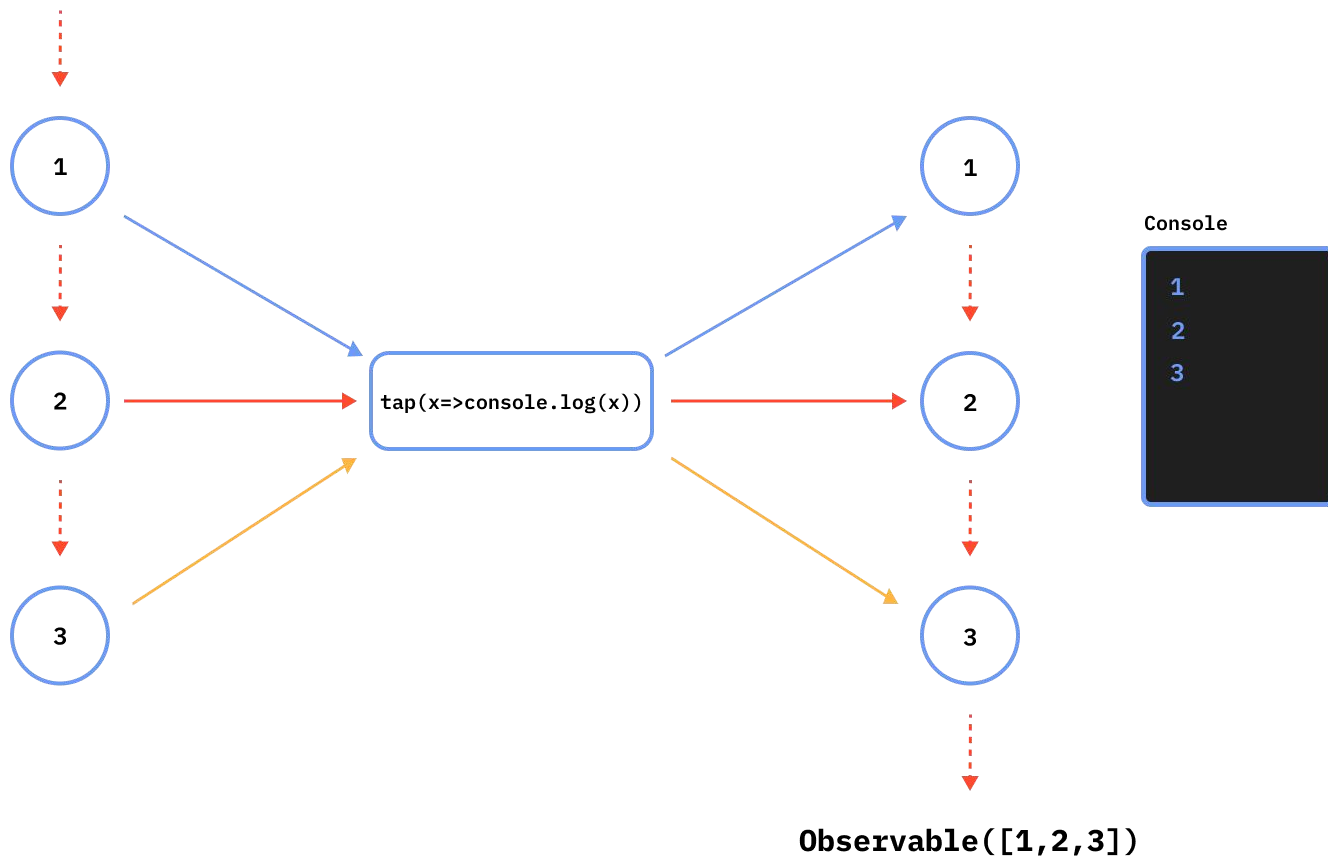
```
of(1, 2, 3)  
  .pipe(filter((x) => x % 2 === 1))  
  .subscribe((nextValue) => console.log(nextValue));
```

tap

Позволява извършване на странични ефекти, като логване, преди елементите да бъдат предадени на следващия оператор или абонат.

Полезно за отстраняване на грешки или действия, които не засягат потока от данни.

Observable([1,2,3])



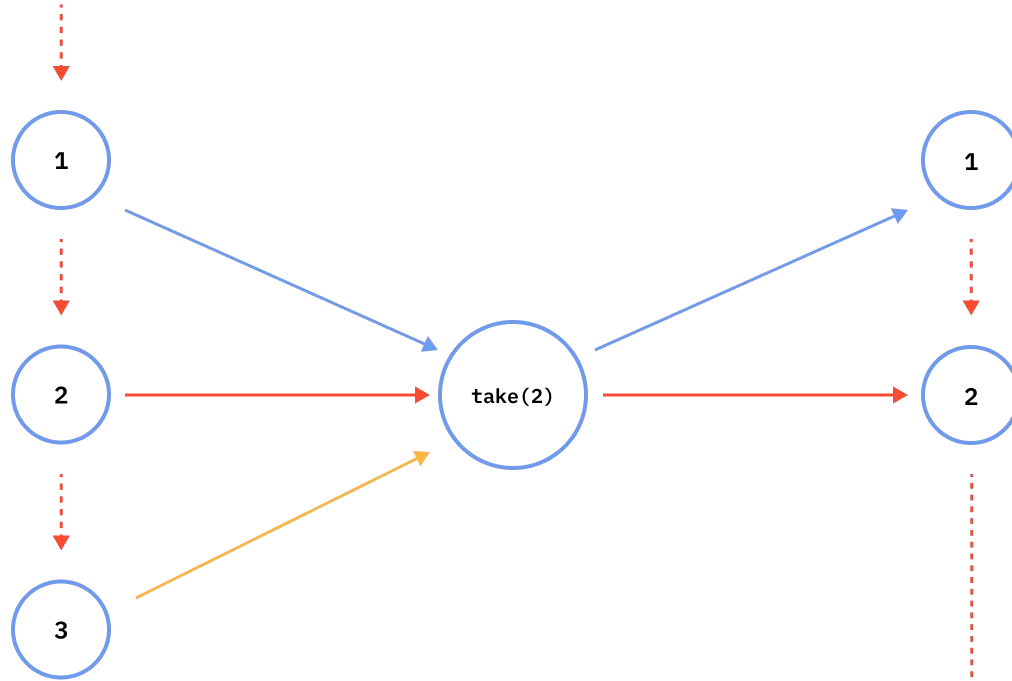
TS app.component.ts

```
of(1, 2, 3)  
  .pipe(tap((x) => console.log('item', x)))  
  .subscribe((nextValue) => console.log(nextValue));
```

take

Ограничава броя на стойностите, които абонатът може да получи от потока. След като са емитирани указаните **n** стойности, потокът автоматично се затваря.

Observable([1,2,3])



Observable([1,2])

TS app.component.ts

```
of(1, 2, 3)  
  .pipe(take(2))  
  .subscribe((nextValue) => console.log(nextValue));
```

Важно!

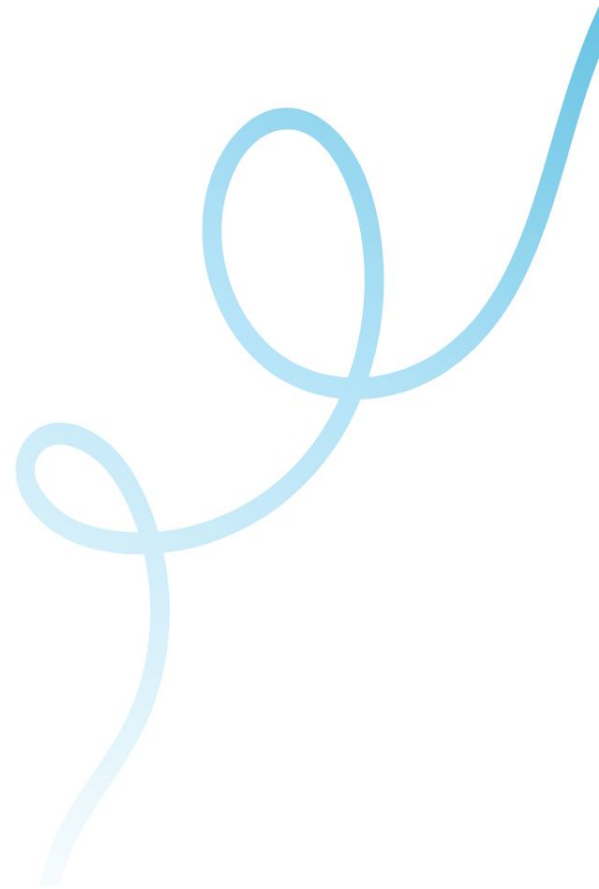
TS app.component.ts

```
of(1, 2, 3, 4, 5)
  .pipe(
    tap((x) => console.log(x)),
    map((x) => x * 2),
    take(2),
    tap((x) => console.log(x))
  )
  .subscribe();
```

Console

1
2
2
4

Упражнение



Работа със свързани данни

Връща нов Observable

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map(x => this.http.get(`some-url/${x}`)),
  )
  .subscribe(data=>console.log(data));
```

Observable

Външен Observable

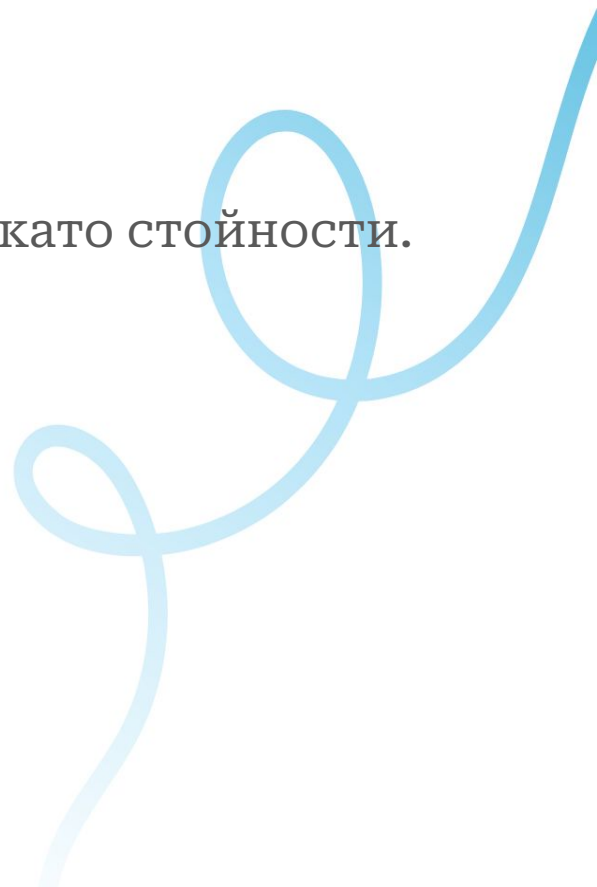
TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map(x => this.http.get(`some-url/${x}`)),
  )
  .subscribe(data=>console.log(data));
```

Вътрешен Observable

higher-order observables

Observables, които излъчват други observables като стойности.



TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map(x => this.http.get(`some-url/${x}`)),
  )
  .subscribe(data=>console.log(data));
```

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map(x => this.http.get(`some-url/${x}`)),
  )
  .subscribe(data=>data.subscribe());
```


Важно!

Влагането на Observable в друг Observable (или т.нар. "callback hell") не се препоръчва, води трудна четимост и загуба на контрол върху потоците.

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map(x => this.http.get(`some-url/${x}`)),
  )
  .subscribe(data=>data.subscribe());
```

higher-order operators

Трансформират стойностите от потока в нови Observables.

Те автоматично създават и премахват абонамента си към нови Observables, като по този начин успяват да “разопаковат” стойностите в тях.

Оператори:

- concatMap
- mergeMap
- switchMap

concatMap

concatMap работи последователно.

Когато външният observable излъчва ново събитие, той изчаква текущият вътрешен observable да завърши, преди да обработи следващия.

Това осигурява гарантиран ред на изпълнение и предпазва от паралелни излъчвания.

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    concatMap((x) => of(x).pipe(delay(1000))),
  )
  .subscribe(data=>console.log(data));
```

Console

1
2
3

mergeMap

mergeMap работи паралелно.

Когато външният observable излъчва ново събитие, той започва нов вътрешен observable веднага, без да изчаква завършването на предишния.

Това означава, че всички вътрешни потоци работят успоредно и резултатите се комбинират при излъчване.

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    mergeMap((x) => of(x).pipe(delay(Math.random() * 1000))),
  )
  .subscribe(data=>console.log(data));
```



Console

```
1
3
2
```

switchMap

switchMap прекъсва текущия вътрешен observable, ако външният observable излъчи ново събитие.

При всяко ново излъчване на външния поток, текущият вътрешен observable се отменя и започва нов вътрешен поток.

Това означава, че се изпълнява само последният стартиран поток.

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    switchMap((x) => of(x).pipe(delay(1000))),
  )
  .subscribe(data=>console.log(data));
```



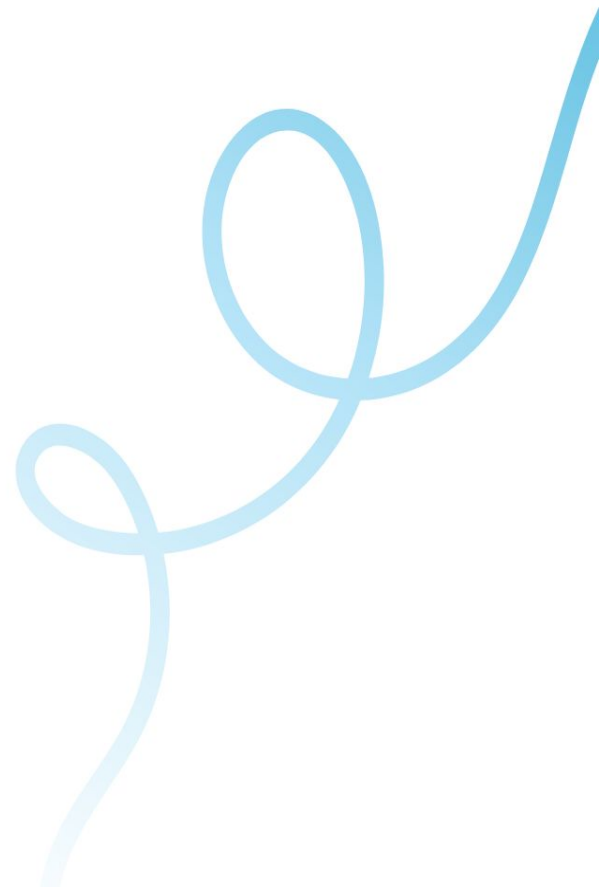
Console

3

Избор на оператор

- **concatMap** - когато искаме последователност и всеки поток да приключи преди да започне следващият нов.
- **mergeMap** - когато искаме паралелно изпълнение и не ни интересува редът на завършване.
- **switchMap** - когато искаме текущите потоци да спрат и да работим само с последния поток, който е стартиран.

Упражнение



Комбиниране на потоци от данни

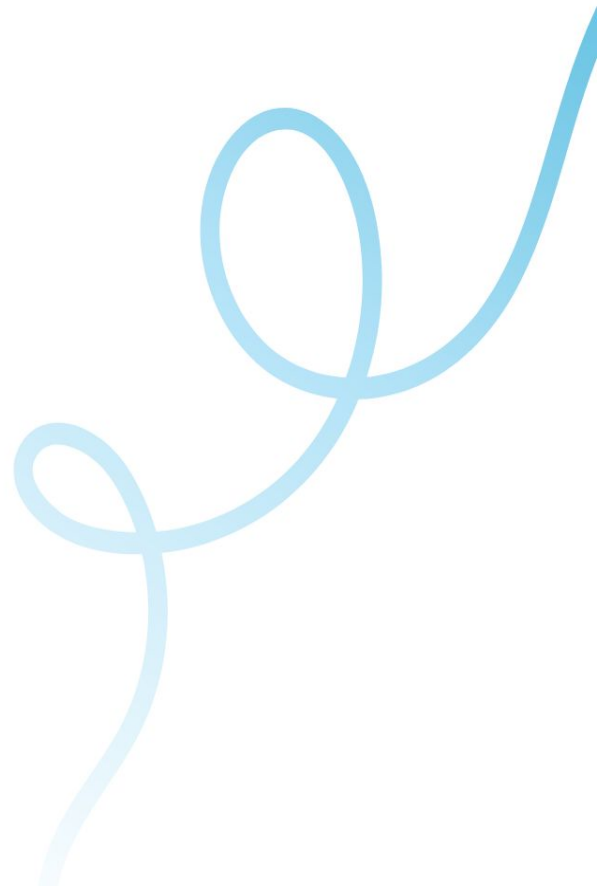
Ползи

- Управление на множество асинхронни източници
- По-лесна синхронизация на данни
- Гъвкавост при обработка на данни



Оператори

- concat
- merge
- forkJoin
- zip
- combineLatest
- withLatestFrom



forkJoin

Този оператор излъчва стойности само когато всички входни observables са завършили.

Полезно, когато искаме да съберем данни от няколко източника и да изчакаме всички операции да приключат, за да използваме резултатите

TS app.component.ts

```
const obs1 = of(1, 2, 3);  
const obs2 = of('a', 'b', 'c');  
  
forkJoin([obs1, obs2]).subscribe(([value1, value2]) =>  
  console.log(value1, value2)  
);
```



Console

3 c

combineLatest

Този оператор комбинира последните стойности от два или повече потока (observables) и връща нов observable, който излъчва масив от последните стойности, когато който и да е от изходните потоци излъчи нова стойност.

TS app.component.ts

```
const obs1 = of(1);  
const obs2 = of('a', 'b', 'c');  
  
combineLatest([obs1, obs2]).subscribe(([value1, value2]) =>  
  console.log(value1, value2)  
);
```



Console

```
1 a  
1 b  
1 c
```

Handling errors and retries

Errors

Грешките в RxJS се третират като крайно събитие в потока.

Когато грешка възникне в даден поток, той прекратява своето изпълнение, освен ако не бъде уловена и обработена чрез специални оператори.

Оператори:

- `throwError`
- `catchError`
- `EMPTY`

throwError

Оператор, който генерира observable, който веднага хвърля грешка когато някой се абонира за него.

Използва се най-често за симулиране на грешки по време на разработка или когато искате умишлено да предадете грешка в потока.

TS app.component.ts

```
const customError$ = throwError(() => 'My error');

customError$.subscribe({
  error: (err) => console.error(err),
});
```

catchError

Оператор за управление на грешки, който позволява да се улавят грешка в поток и да бъде изпълнена алтернативна логика.

Когато възникне грешка, той замества потока с нов.

TS app.component.ts

```
const source$ = throwError(() => 'Грешка!'); // Симулира поток с грешка

source$
  .pipe(
    catchError((err) => {
      return of('Резервна стойност'); // Връщане на алтернативен поток при грешка
    })
  )
  .subscribe({
    next: (value) => console.log(value), // Output: 'Резервна стойност'
    error: (err) => console.error('Грешка:', err),
    complete: () => console.log('Потокът завърши'),
  });
```

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map((item) => {
      if (item === 2) {
        throw 'Error';
      }

      return item;
    }),
    catchError((err) => {
      console.error(err);
      return of('Резервна стойност');
    })
  )
  .subscribe({
    next: (value) => console.log(value),
    error: (err) => console.error('Грешка:', err),
    complete: () => console.log('Потокът завърши'),
  });
```

Console

1
Резервна стойност
Потокът завърши

EMPTY

Създаващ оператор, който генерира празен поток.

Това е поток, който не няма никакви стойности и завършва веднага.

Може да бъде полезен, когато искаме поток, който да не изпълнява никакви действия, но да завърши без грешка.

TS app.component.ts

```
const empty$ = EMPTY; // Празен поток

empty$.subscribe({
  next: (value) =>
    console.log('Тази стойност никога няма да бъде емитирана'),
  complete: () => console.log('Потокът завърши'), // Output: 'Потокът завърши'
});
```

Retries

RxJS предоставя възможност за възстановяването на потоци в случаи на грешки.

Те са изключително полезни за ситуации с временни проблеми (напр. мрежови грешки), когато искаме да опитаме да повторим операцията няколко пъти преди да се откажем или да обработим грешката по друг начин.

Оператор:

- `retry`

retry

Оператор, който автоматично рестартира изпълнението на поток в случай на грешка.

Това означава, че ако потокът срещне грешка, `retry` ще накара потока да започне отначало, без да се налага ръчна намеса

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map((item) => {
      if (item === 2) {
        throw 'Error';
      }

      return item;
    }),
    retry(3)
  )
  .subscribe({
    next: (value) => console.log(value),
    error: (err) => console.error('Грешка:', err),
    complete: () => console.log('Потокът завърши'),
  });
```

Важно!

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map((item) => {
      if (item === 2) {
        throw 'Error';
      }

      return item;
    }),
    catchError((err) => {
      return of('Грешка');
    }),
    retry(3)
  )
  .subscribe({
    next: (value) => console.log(value),
    error: (err) => console.error('Грешка:', err),
    complete: () => console.log('Потокът завърши'),
  });
```

Персонализирана логика

Retry операторът ни предоставя начин, по който ние можем сами да определим логиката при рестартирането на потока.

```
interface RetryConfig {  
  count?: number  
  delay?: number | ((error: any, retryCount: number) => ObservableInput<any>)  
  resetOnSuccess?: boolean  
}
```

TS app.component.ts

```
of(1, 2, 3)
  .pipe(
    map((item) => {
      if (item === 2) {
        throw 'Error';
      }

      return item;
    }),
    retry({
      count: 5,
      delay: (error, retryCount) => {
        return retryCount === 1 ? timer(2000) : of({});
      },
    })
  )
  .subscribe();
```


Благодаря за вниманието!