

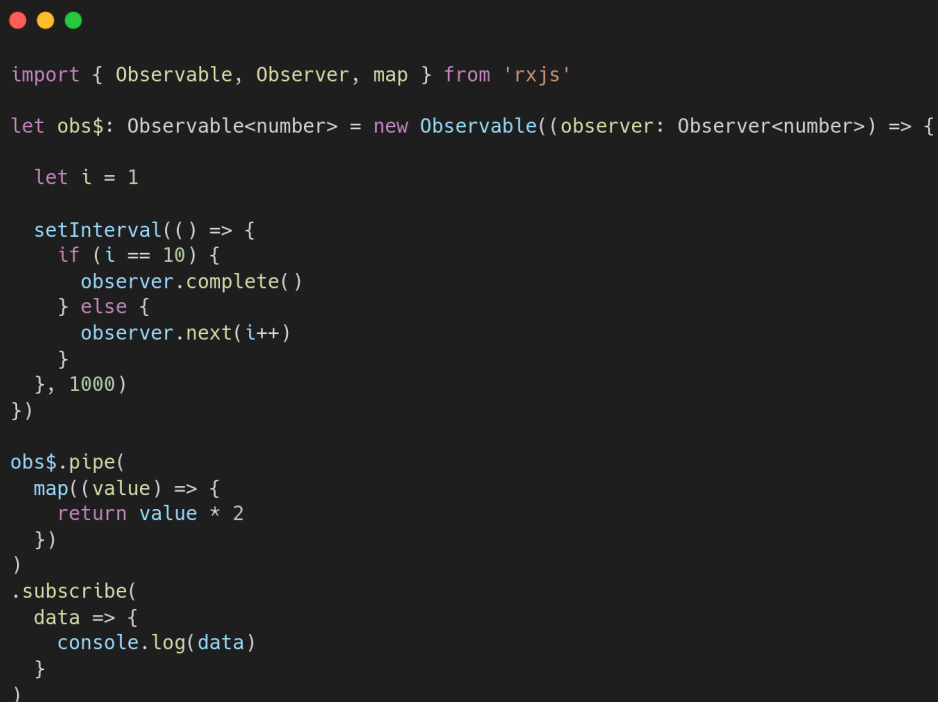
# Operadores RXJS

Os operadores do RXJS são, nada mais, do que funções que podem ser executadas no seu observable para processar os dados de alguma maneira. Nós podemos filtrar dados, remover dados, mapear dados, etc... Basta apenas utilizar seus operadores e podemos manipular os dados retornados como bem entendermos. Para utilizarmos os operadores, basta que, antes de dar o subscribe no observable, utilizar o método **pipe()** e, dentro dele, você informa quais os operadores deseja utilizar. Veremos isso melhor em breve.

Vamos ver alguns exemplos importantes de operadores que podemos utilizar:

## Map

Assim como o método map dos Arrays, o operador map recebe uma função que faz modificações nos dados e os retornam com as devidas alterações.

A screenshot of a code editor with a dark background and light-colored text. The code is in TypeScript and demonstrates the use of the RxJS 'map' operator. It starts with an import statement for 'Observable', 'Observer', and 'map' from 'rxjs'. Then, it creates an Observable named 'obs\$' that emits numbers from 1 to 10 using 'setInterval'. The 'pipe' method is used to chain the 'map' operator, which multiplies each value by 2. Finally, the 'subscribe' method is called to log the transformed values to the console.

```
import { Observable, Observer, map } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  map((value) => {
    return value * 2
  })
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

Executando esse código, você verá que os números serão retornados com seu dobro. Quando o valor informado for **1**, o valor mostrado será **2**, pois é seu dobro. E assim irá até o fim de sua execução.

## Filter

O operador filter, assim como o método filter dos arrays, filtra dados a partir de uma condição. Vamos fazer com que seja retornado apenas os números pares

```
import { Observable, Observer, filter } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  filter(value => {
    return value % 2 == 0
  })
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

Veja que, ao mostrar no console, aparecerá apenas os números pares.

## Take

O operador take nos permite informar quantos valores da fonte de dados nós queremos recuperar, permitindo que peguemos somente quanto forem precisos.

```
import { Observable, Observer, take } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  take(3)
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

com esse código, somente os três primeiros valores serão retornados.

## First

Assim como no take, nós utilizamos o first para pegar uma certa quantidade de elementos. Mas, nesse caso, ele pegará somente o primeiro dado retornado. Todos os outros serão ignorados.

```
import { Observable, Observer, first } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  first()
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

Assim, ele retornará somente o primeiro dado que for retornado. Mas ele também possui outra maneira de uso. Nós podemos fazer com que ele retorne o primeiro elemento que satisfaça uma condição. Por exemplo, nós podemos pegar o primeiro elemento que seja par. Basta apenas passar uma função para ele informando isso

```
import { first, Observable, Observer, take } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  first((x) => {
    return x % 2 == 0
  })
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

Assim, quando ele achar o primeiro elemento que satisfaça essa condição ele irá retorná-lo e o resto será ignorado.

## Last

Ao contrário do first, ele retornará somente o último dado que for retornado da fonte de dados. Todos os outros serão ignorados

```
import { Observable, Observer, last } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  last()
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

Dessa maneira, somente o último dado será retornado. Mas, assim como o `first`, podemos passar uma função com uma condição. Caso essa condição for atendida e ele for o último dado que atende essa condição, ele será retornado.

```
import { last, Observable, Observer } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  last(x => x % 2 == 0)
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

Nesse caso, ele retornará o último que seja par

## TakeWhile

O TakeWhile também recupera uma determinada quantidade de elementos, mas ele retornará enquanto uma determinada condição for verdadeira. Vamos fazer com que ele retorne os números enquanto eles forem menores ou iguais que 5

```
import { last, Observable, Observer, takeWhile } from 'rxjs'

let obs$: Observable<number> = new Observable((observer: Observer<number>) => {

  let i = 1

  setInterval(() => {
    if (i == 10) {
      observer.complete()
    } else {
      observer.next(i++)
    }
  }, 1000)
})

obs$.pipe(
  takeWhile((x) => x <= 5)
)
.subscribe(
  data => {
    console.log(data)
  }
)
```

Enquanto essa condição for verdadeira, ele retornará os dados

## DebounceTime

Esse operador faz com que seja esperando um determinado tempo antes que o subscribe seja executado e os dados sejam retornados. Para entendermos melhor, vamos levar em consideração o seguinte cenário: Vamos considerar que você tem um input de pesquisa no seu site e, a cada digitação, você fará uma requisição a uma API. Se você fizer a requisição a cada nova letra que o usuário digitar dentro do seu input, ele pode fazer dezenas, ou até centenas, de requisições, deixando sua aplicação muito lenta e difícil de utilizar.

Mas, para contornar isso, podemos utilizar uma técnica chamada **debounce**. Essa técnica consiste em esperar um tempo X para fazer alguma coisa a cada interação do usuário com algum elemento. Utilizando o operador debounceTime, nós podemos fazer esse trabalho de uma maneira ágil. Vamos criar um exemplo de como isso funcionaria. Vamos criar um input no nosso template HTML.



```
<input
  type="text"
  placeholder="Type something"
>
```

Agora, vamos criar um Subject que retornará strings.

```
import { Component } from '@angular/core';
import { Subject } from 'rxjs';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  debounceSub$: Subject<string> = new Subject()
}
```

Agora, vamos ouvir o evento de input no nosso input. Esse evento vai executar um método chamado **debounceTest()** que iremos criar em breve

```
<input
  type="text"
  placeholder="Type something"
  (input)="debounceTest($event)"
>
```

Agora vamos criar esse método. Esse método chamará o nosso subject e dará um **next()** no conteúdo que foi digitado no input

```
import { Component } from '@angular/core';
import { Subject } from 'rxjs';
import { debounceTime } from 'rxjs/operators';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  debounceSub$: Subject<string> = new Subject()

  debounceTest(event: any) {
    this.debounceSub$.next(event.target.value)
  }
}
```

Agora, precisamos também dar um subscribe nesse subject. Vamos utilizar o lifecycle **ngOnInit** para fazer isso e vamos passar o nosso debounceTime com um tempo de 2 segundos.

```
import { Component } from '@angular/core';
import { Subject } from 'rxjs';
import { debounceTime } from 'rxjs/operators';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  debounceSub$: Subject<string> = new Subject()

  ngOnInit() {
    this.debounceSub$
      .pipe(
        debounceTime(2000)
      )
      .subscribe(
        data => {
          console.log('You typed', data)
        }
      )
  }

  debounceTest(event: any) {
    this.debounceSub$.next(event.target.value)
  }
}
```

Agora digite algo no seu input e veja que ele não escreverá no console sempre. Ele apenas digitará sempre que você parar de digitar por 2 segundos. Se caso, durante esse tempo, você continuar digitando, nada aparecerá no console.