Czym jest RxJs?

- RxJS to biblioteka (JavaScript) do programowania reaktywnego, czyli asynchronicznego programowania związanego z wykorzystaniem obserwowalnych strumieni danych. RxJS dostarcza typ tzw. **Observable**.
- Angular używa biblioteki RxJS do m.in.: obsługi żądań HTTP (zwraca tzw. Observable), w formularzach, obsługi routingu oraz przy EventEmitter (dziedziczy po klasie Subject, który jest Observable).

```
export class TestService {
  no usages
  constructor(private http: HttpClient) {}
  no usages
  getNames(): Observable<string[]> {
    return this.http.get<string[]>( url: 'url');
  }
}
```

```
export class AppComponent implements OnInit{
  no usages
  constructor(private testService: TestService) {}
  no usages
  ngOnInit(): void {
    this.testService.getNames().subscribe( observerOrNext: {
      next: value => {
        console.log(value);
      },
      error: err => console.error(err),
      complete: () => console.log('Zakończono!')
    })
  }
}
```

Czym jest Observable i Observer?

- **Observable** – to obiekt, który emituje wartości przekazywane asynchronicznie (strumień danych). Możemy się do niego subskrybować za pomocą metody **subscribe()** i nasłuchiwać na pojawiające się wartości. Kiedy subskrybujemy się do Observable – to zwraca nam obiekt **Subscription** (możemy przypisać go do zmiennej i np. użyć w OnDestroy do odsubskrybowania).
- **Observer** – to obiekt, nasłuchuje na wartości emitowane przez Observable. Można go rozumieć jako odbiorcę danych, który w jakiś sposób reaguje na pojawiające się wartości. Obiekt **Observer** składa się z trzech funkcji – **next()** (kiedy Observable emituje nową wartość), **error()** (Observable zwraca błąd), **complete()** (kiedy Observable kończy emisję wartości). Obiekt Observer przekazujemy do metody **subscribe()**.

Tworzenie Observera

- Zamiast do ciała metody **subscribe**() przekazywać nawiasy klamrowe (obiekt Observer) i tam reagować na **next, error** i **complete** możemy utworzyć **Observer** osobno, dzięki czemu będziemy mogli go stosować w kilku miejscach (komponentach itd.)

```
export class TestService {
  no usages
  constructor(private http: HttpClient) {}
  no usages
  getNames(): Observable<string[]> {
    return this.http.get<string[]>( url: 'url');
  }
}
```

```
export class AppComponent implements OnInit{
  no usages
  constructor(private testService: TestService) {}
  no usages
  ngOnInit(): void {
    this.testService.getNames().subscribe(this.myObserver)
  }
  1 usage
  myObserver: Observer<string[]> = {
    next: value => console.log(value),
    error: err => console.error('Wystąpił błąd:', err),
    complete: () => console.log('Emisja wartości zakończona'),
  };
}
```

Observable, czy Promise?

Promise są wbudowane w język JavaScript i są wykorzystywane do asynchronicznego działania kodu. Jednak Observable dają większe możliwości, jeśli chodzi o pracę z asynchronicznym kodem.

Promise:

- Jest **eager**, tzn. że wykonuje się od razu.
- Dostarcza jedną wartość.
- Nie można przerwać jego działania.
- Nie posiada wbudowanych operatorów, łączenie za pomocą **.then()**.
- Obsługa błędów poprzez **.catch()**.

Observable:

- Jest **lazy**, tzn. wykonuje się dopiero po subskrypcji.
- Może dostarczać strumień z wieloma wartościami.
- Można przerwać jego działanie poprzez odsubskrybowanie.
- Posiada wbudowane operatory do wpływania na dane przychodzące ze strumienia danych.
- Dedykowane operatory do obsługi błędów.

Czym jest Subject?

- **Subject** – to rodzaj Observable, które działa również jako Observer. Można go rozumieć jako połączenie strumienia danych i odbiorcy, który może jednocześnie emitować wartości (metoda **next()**) i nasłuchiwać (subskrybować) na wartości. Nie przyjmuje żadnych danych wejściowych. Posiada kilka odmian (m.in. **BehaviorSubject** – który działa tak samo oprócz tego, że przyjmuje dane wejściowe oraz przechowuje ostatnio emitowaną wartość).

```
export class TestService {
  1 usage
  test$ = new Subject<string>();
  1 usage
  name!: string;
  no usages
  constructor() {}
  1 usage
  changeName(newName: string): void {
    this.name = newName;
    this.test$.next(newName);
  }
}
```

```
export class AppComponent implements OnInit{
  no usages
  constructor(private testService: TestService) {
  }
  no usages
  ngOnInit(): void {
    //od Subject trzeba odsubskrybować się!(kolejne slajdy)
    this.testService.test$.subscribe( observerOrNext: {
      next: name => {
        console.log(name)
      }
    })
    this.testService.changeName( test: 'test');
  }
}
```

W powyższym kodzie w metodzie **changeName()** za każdym razem, gdy właściwość **name** otrzyma nową wartość emitujemy tą wartość poprzez **Subject**. Dzięki temu możemy subskrybować się do tego **Subject** w różnych komponentach i reagować na zmianę **name**.

Kiedy trzeba użyć **unsubscribe()**, a kiedy nie?

- Do Observable możemy się subskrybować (metoda **subscribe()**), ale i odsubskbrybować (metoda **unsubscribe()**). W niektórych sytuacjach np. kiedy pracujemy ze skończonymi strumieniami danych nie ma konieczności używania unsubscribe().

Trzeba:

- Subskrybujemy się do **Subject**/**BehaviorSubject**.
- Subskrybujemy się do eventów DOM za pomocą **fromEvent()**.
- Subskrypcja do Store w **NgRx**.
- Nieskończone strumienie danych – np. metoda **interval()**.
- Formularze – metoda **valueChanges()**, czyli subskrypcja do kontrolki formularza.

Nie trzeba:

- Używamy skończonego strumienia **of()**.
- Subskrybujemy się do Observable z HttpClient (żadania HTTP).
- Subskrybujemy się do Observable z Routera.
- Po wykorzystaniu **AsyncPipe**.

Jak użyć unsubscribe()?

Bardzo ważne jest żeby używać unsubscribe(), wtedy kiedy jest to konieczne! Inaczej po zniszczeniu komponentu subskrypcja dalej będzie działać, co może spowodować bardzo duże wycieki pamięci i problemy z działaniem aplikacji.

```
export class TestService {
  1 usage
  test$ = new Subject<string>();
```

```
export class AppComponent implements OnInit, OnDestroy{
  2 usages
  sub!: Subscription;
  no usages
  constructor(private testService: TestService) {}
  no usages
  ngOnInit(): void {
    this.sub = this.testService.test$.subscribe( observerOrNext: {
      next: name => console.log(name)
    })
  }
  no usages
  ngOnDestroy(): void {
    this.sub.unsubscribe();
  }
}
```

Alternatywny sposób na unsubscribe()

```
export class AppComponent implements OnInit, OnDestroy
  private subs = new Subscription();
  no usages
  constructor(private testService: TestService) {}
  no usages
  ngOnInit(): void {
    this.subs.add(
      this.testService.test$.subscribe( observerOrNext: {
        next: value => console.log('value')
      })
    )
  }
  no usages
  ngOnDestroy(): void {
    this.subs.unsubscribe();
  }
}
```

Tworzenie własnych Observable (operatory tworzące)

RxJS udostępnia funkcję do tworzenia własnych Observable:

- **of()**
- **from()**
- **interval()**
- **timer()**
- **fromEvent()**
- **defer()**

```
import {interval, Observable, of, timer, from} from "rxjs";
```

```
obs = of( value: [1, 2, 3]).subscribe( observerOrNext: {
  next: (numb : number[] ) => console.log(numb)
})
```

Emitowanie jednej wartości (tablicy z liczbami).

Operatory RxJS

- Operatory w RxJS to funkcje, które pozwalają na transformację, filtrowanie i manipulowanie strumieniami danych w czasie rzeczywistym.
- Używanie operatorów jest dostępne dzięki metodzie **.pipe()**.
- Jest podział zależnie od tego, co wykonuje dany operator. Istnieją operatory kombinacyjne, filtrujące, operatory matematyczne, operatory transformujące itd.
- Oprócz wbudowanych operatorów RxJS można tworzyć także własne, ale istnieje duża szansa że biblioteka RxJS już zawiera operator, który będzie nam potrzebny.

```
obs = of( values: 1,  values: 2,  values: 3) ...
  .pipe(map( project: numb => numb * 2)) Observable<number>
  .subscribe( observerOrNext: {
  next: (numb : number ) => console.log(numb)
})
```