RxJS (Reactive Extensions for JavaScript)

RxJS (Reactive Extensions for JavaScript) is a powerful library for handling asynchronous data streams in Angular and JavaScript applications.

Core RxJS Concepts

In **RxJS**, everything revolves around **Observables**. An **Observable** is a stream of data that can be observed over time.

1 **Observable** → Represents a data stream (like a TV channel).
2 **Observer** → Listens for emitted values (like a person watching TV).
3 **Subscribe** → Connects the observer to the observable (like turning on the TV).

1 Observable - Creating a Data Stream

An **Observable** is a function that produces values over time. You create an Observable using `new Observable()` or operators like `of()`, `interval()`, etc.

Example: Creating an Observable

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

// Creating an Observable
const myObservable = new Observable(observer => {
  observer.next('Hello');
  observer.next('World');
  observer.complete();
});

// Subscribing to Observable
myObservable.subscribe(value => console.log(value));
```

Output:

```
Hello
World
```

2 Observer - Reacting to Data

An **Observer** is an object that defines how to react to values emitted by the Observable. It has three methods:

- **`next(value)`** → Handles incoming values.
- **`error(err)`** → Handles errors.
- **`complete()`** → Called when the Observable finishes.

Example: Using an Observer

```
observer = {
    next: (value: any) => console.log('Received:', value),
    error: (err: any) => console.error('Error:', err),
    complete: () => console.log('Done!')
  };

  observable = new Observable<number>(obs => {
    obs.next(1);
    obs.next(2);
    obs.complete();
  });

  constructor() {
    // Subscribe with an observer
    this.observable.subscribe(this.observer);
  }
```

Output:

```
Received: 1
Received: 2
Done!
```

3 Subscribe - Connecting Observer to Observable

The `subscribe()` method **connects** an observer to an observable. Without subscribing, an Observable **does nothing**.

Example: Manually Subscribing

```
observable = new Observable(observer => {
    setTimeout(() => observer.next('Data 1'), 1000);
    setTimeout(() => observer.next('Data 2'), 2000);
    setTimeout(() => observer.complete(), 3000);
  });

   subscription = this.observable.subscribe(
    value => console.log('Received:', value),
    error => console.error('Error:', error),
    () => console.log('Observable Complete')
  );
```

Output (with delays):

```
 (After 1 sec) Received: Data 1
(After 2 sec) Received: Data 2
(After 3 sec) Observable Complete
```

Unsubscribing from an Observable

To stop receiving values, **unsubscribe** from the observable.

```
 observable = new Observable<string>(observer => {
    setTimeout(() => observer.next('Data 1'), 1000);
    setTimeout(() => observer.next('Data 2'), 2000);
    setTimeout(() => observer.complete(), 3000);
  });

  subscription: any; // To store the subscription reference

  constructor() {
    // Subscribe to the observable
    this.subscription = this.observable.subscribe({
      next: value => console.log(value),
      complete: () => console.log('Observable completed'),
    });

    // Unsubscribe after 2 seconds
    setTimeout(() => {
      this.subscription.unsubscribe();
      console.log('Unsubscribed');
    }, 2000);
  }
```

Key Points

1 **Observable** emits data over time.
2 **Observer** defines how to handle the data.
3 **Subscribe** starts the data flow.
4**Unsubscribe** stops listening.

Real-World example

```
class AppComponent implements OnInit, OnDestroy {

  // Observable: Simulate order status updates
  orderStatusObservable=new  Observable<string>;

  // Subscription to manage the observable stream
  subscription: any;

  // Observer: Customer receiving updates about the order status
  customerObserver = {
    next: (status: string) => console.log('Order Status:', status),
    error: (err: any) => console.error('Error:', err),
    complete: () => console.log('Order tracking completed!')
  };

  // ngOnInit lifecycle method to handle observable creation and subscription
  ngOnInit() {
    this.orderStatusObservable = new Observable(observer => {
      // Emit order statuses at different times
      observer.next('Order Placed');   // First status
      setTimeout(() => observer.next('Shipped'), 2000);    // After 2 seconds
      setTimeout(() => observer.next('Out for Delivery'), 4000); // After 4
seconds
      setTimeout(() => observer.next('Delivered'), 6000);  // After 6 seconds

      // Complete the stream after the last update
      setTimeout(() => observer.complete(), 6000);
    });

    // Customer subscribes to the order status updates
    this.subscription =
this.orderStatusObservable.subscribe(this.customerObserver);
  }

  // Optionally unsubscribe after a certain period if no longer interested
  ngOnDestroy() {
    // Unsubscribe after 5 seconds
    setTimeout(() => {
      this.subscription.unsubscribe();
      console.log('Unsubscribed from order status updates');
    }, 5000);
  }
}
```

Subject

In **RxJS**, a **`Subject`** is a type of **Observable** that allows values to be multicasted to many Observers. Unlike a regular Observable, a **Subject** can act as both an **Observer** and an **Observable**.

In simpler terms:

- A **regular Observable** only emits values to its subscribers.
- A **Subject** allows values to be **multicast** (sent to multiple subscribers) and also allows you to **push values** to the stream (just like an Observer).

Types of Subjects in RxJS

There are a few variations of the `Subject` in RxJS, each with different behavior:

1. **Basic Subject**
2. **BehaviorSubject**
3. **ReplaySubject**
4. **AsyncSubject**

1. Basic Subject

A basic `Subject` works as a simple multicast mechanism. It can **emit values** to all its subscribers at once. It doesn't store any values, meaning if you subscribe after a value is emitted, you will not receive any previous values.

Example:

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

subject = new Subject<number>(); // Initialize the Subject

  constructor() {
    // Subscribe observers
    this.subject.subscribe((value: number) => console.log('Observer 1:', 
value));
    this.subject.subscribe((value: number) => console.log('Observer 2:', 
value));

    // Emit values using the subject instance
    this.subject.next(1);
    this.subject.next(2);
}
```

 Output:

```
Observer 1: 1
Observer 2: 1
Observer 1: 2
Observer 2: 2
```

- When `subject.next(1)` is called, **both subscribers** receive the value `1`.
- When `subject.next(2)` is called, **both subscribers** again receive the value `2`.

2. BehaviorSubject

A `BehaviorSubject` is like a `Subject`, but with one key difference: it **holds the latest value** and **sends it to new subscribers immediately** upon subscription. You specify an initial value when creating a `BehaviorSubject`, and if a new subscriber subscribes later, it will immediately get the **latest emitted value**.

Example:

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

// Create a new BehaviorSubject with an initial value
```

```
  behaviorSubject = new BehaviorSubject<number>(0); // Initialize the Subject

  constructor() {
    // Subscribe observers
    this.behaviorSubject.subscribe(value => console.log(`Subscriber 1
received: ${value}`));
    this. behaviorSubject.next(1);
    this.behaviorSubject.next(2);

// Subscriber 2 (subscribes after some values are emitted)
this.behaviorSubject.subscribe(value => console.log(`Subscriber 2 received:
${value}`));

// Emit more values
this.behaviorSubject.next(3);
}
```

Output:

```
Subscriber 1 received: 0
Subscriber 1 received: 1
Subscriber 1 received: 2
Subscriber 2 received: 2
Subscriber 1 received: 3
Subscriber 2 received: 3
```

- **Subscriber 1** receives all emitted values, including the initial value `0`.
- **Subscriber 2** receives the **latest value `2`** upon subscription (because `BehaviorSubject` stores the latest value).

3. ReplaySubject

A `ReplaySubject` is similar to a `Subject`, but it **replays a set number of previous values** to new subscribers. When a new subscriber subscribes, the `ReplaySubject` will replay the last **N emitted values**, where `N` is the number of values you configure it to store.

Example:

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

// Create a new ReplaySubject that stores the last 2 emitted values
const replaySubject = new ReplaySubject<number>(2);

// Subscriber 1
replaySubject.subscribe(value => console.log(`Subscriber 1 received:
${value}`));

// Emit values
replaySubject.next(1);
replaySubject.next(2);
replaySubject.next(3);

// Subscriber 2 (subscribes after some values are emitted)
```

```
replaySubject.subscribe(value => console.log(`Subscriber 2 received: 
${value}`));
```

Output:

```
Subscriber 1 received: 1
Subscriber 1 received: 2
Subscriber 1 received: 3
Subscriber 2 received: 2
Subscriber 2 received: 3
```

- **Subscriber 2** receives the **last two values `2` and `3`** because we configured the `ReplaySubject` to replay the last 2 emitted values.

4. AsyncSubject

An `AsyncSubject` only **emits the last value** when the **observable completes**. Unlike other subjects, it only emits a value when `complete()` is called. If a subscriber subscribes before the `complete()` method is called, they will receive nothing until the observable completes.

Example:

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

// Create a new AsyncSubject
const asyncSubject = new AsyncSubject<number>();

// Subscriber 1
asyncSubject.subscribe(value => console.log(`Subscriber 1 received: 
${value}`));

// Emit values
asyncSubject.next(1);
asyncSubject.next(2);
asyncSubject.next(3);

// Complete the observable, so values will be emitted
asyncSubject.complete();
```

Output:

```
Subscriber 1 received: 3
```

- **Subscriber 1** only receives the **last emitted value** (`3`) after `complete()` is called.

Key Differences Between Subject Types

Type	Behavior	Use Case
Subject	Emits values to all subscribers at the time of emission.	Use for multicasting values in real-time to multiple observers.

Type	Behavior	Use Case
BehaviorSubject	Holds the latest value and sends it to new subscribers immediately.	Use when you want new subscribers to receive the most recent value.
ReplaySubject	Replays the last N values to new subscribers.	Use when you want new subscribers to receive the most recent N values.
AsyncSubject	Only emits the last value when the observable completes.	Use when you want to send only the last emitted value after completion.

Conclusion

- A **`Subject`** allows you to multicast and **push values** to multiple subscribers.
- **`BehaviorSubject`** keeps track of the most recent value and gives it to new subscribers.
- **`ReplaySubject`** replays the last N values to new subscribers.
- **`AsyncSubject`** only emits the last value when the observable is complete.

`Subject` is powerful when you need to manage streams of data and broadcast to multiple parts of your application.

Operators (Pipeable Functions)

RxJS **operators** are **pure functions** that allow you to transform, filter, combine, and manage asynchronous data streams in Observables. These functions are used inside `.pipe()`.

Types of RxJS Operators

1. Creation Operators – Create new Observables

2. Transformation Operators – Modify emitted values

3. Filtering Operators – Filter emitted values

4. Combination Operators – Combine multiple Observables

5. Error Handling Operators – Handle errors gracefully

6. Utility Operators – Perform side effects or timing operations

1 Creation Operators

Used to create new Observables.

Operator	Description	Example
`of`	Emits provided values sequentially	`of(1, 2, 3).subscribe(console.log);`
`from`	Converts an array, promise, or iterable into an Observable	`from([10, 20, 30]).subscribe(console.log);`
`interval`	Emits values at specified time intervals	`interval(1000).subscribe(console.log);`
`timer`	Emits a value after a delay, then optionally continues emitting	`timer(2000).subscribe(() => console.log('After 2s'));`
`range`	Emits a range of numbers sequentially	`range(1, 5).subscribe(console.log);`

2 Transformation Operators

Used to modify emitted values.

Operator	Description	Example
`map`	Transforms each emitted value	`of(2, 4, 6).pipe(map(x => x * 2)).subscribe(console.log);`
`pluck`	Extracts a specific property from emitted objects	`from([{name: 'Alice'}, {name: 'Bob'}]).pipe(pluck('name')).subscribe(console.log);`
`bufferTime`	Collects values for a specific time period and emits as an array	`interval(1000).pipe(bufferTime(3000)).subscribe(console.log);`

3 Filtering Operators

Used to filter out unwanted values.

Operator	Description	Example
`filter`	Only emits values that pass a condition	`of(1, 2, 3, 4, 5).pipe(filter(x => x % 2 === 0)).subscribe(console.log);`
`take`	Takes the first N values and then completes	`interval(1000).pipe(take(3)).subscribe(console.log);`
`first`	Emits only the first value that matches a condition	`of(10, 20, 30).pipe(first(x => x > 15)).subscribe(console.log);`
`skip`	Skips the first N values	`of(1, 2, 3, 4, 5).pipe(skip(2)).subscribe(console.log);`
`distinct`	Removes duplicate values	`of(1, 2, 2, 3, 3, 4).pipe(distinct()).subscribe(console.log);`

4 Combination Operators

Used to combine multiple Observables.

Operator	Description	Example
`Merge`	Merges multiple Observables	`merge(of(1, 2), of(3, 4)).subscribe(console.log);`
`Concat`	Runs Observables sequentially	`concat(of(1, 2), of(3, 4)).subscribe(console.log);`
`combineLatest`	Emits combined latest values from multiple Observables	`combineLatest([of(1, 2), interval(1000)]).subscribe(console.log);`

Operator	Description	Example
`Zip`	Combines multiple Observables like a zipper	`zip(of(1, 2), of('A', 'B')).subscribe(console.log);`

5 Error Handling Operators

Used to handle errors in streams.

Operator	Description	Example
`catchError`	Catches an error and returns a fallback value	`throwError(() => new Error('Oops!')).pipe(catchError(() => of('Recovered'))).subscribe(console.log);`
`Retry`	Retries an Observable N times before failing	`of('Error').pipe(map(() => { throw new Error('Retrying...'); }), retry(2)).subscribe(console.log);`

---

6 Utility Operators

Used for debugging, side effects, and timing.

Operator	Description	Example
`tap`	Performs side effects like logging without modifying values	`of(1, 2, 3).pipe(tap(x => console.log('Before:', x)), map(x => x * 10)).subscribe(console.log);`
`delay`	Delays each emission by a given time	`of('Hello').pipe(delay(2000)).subscribe(console.log);`
`finalize`	Runs a function when the Observable completes	`of('Complete').pipe(finalize(() => console.log('Finished'))).subscribe(console.log);`

Example: **map() operator** - transforms values.

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

of(2, 4, 6)
  .pipe(map(value => value * 2))
  .subscribe(result => console.log(result));
```

Output:

```
4
8
12
```

Step 1: `of(2, 4, 6)`

- The `of` operator **creates an Observable** that emits the values **2, 4, and 6** sequentially.
- This means that an **observable stream** is created that will **emit these values one by one** and then complete.

Equivalent to:

```
const observable = new Observable(observer => {
  observer.next(2);
  observer.next(4);
  observer.next(6);
  observer.complete();
});
```

Step 2: `.pipe(map(value => value * 2))`

- The `.pipe()` method is used to **apply operators** to the emitted values.
- The `map` operator **transforms each emitted value** by multiplying it by 2.
- Each value passes through the `map` function and gets transformed.

Transformation logic:

- 2 → 2 * 2 = **4**
- 4 → 4 * 2 = **8**
- 6 → 6 * 2 = **12**

Step 3: `.subscribe(result => console.log(result))`

- The `.subscribe()` method **listens** to the Observable and executes the callback function whenever a value is emitted.

- Here, it simply logs each transformed value.

A. Creation Operators

1. `of()` - Emits values in sequence

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

of(1, 2, 3, 4).subscribe(value => console.log(value));
```

Output:

```
1
2
3
4
```

2. `range()` - Emits a range of numbers

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

range(1, 5).subscribe(console.log);
```

Output:

```
1
2
3
4
5
```

B. Transformation Operators

3. `map()` - Transforms values

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

of(2, 4, 6)
  .pipe(map(value => value * 2))
  .subscribe(result => console.log(result));
```

Output:

```
4
8
12
```

4. `concatMap()` - Maintains order and waits for each inner Observable to complete

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

of('A', 'B', 'C')
  .pipe(concatMap(value => of(value).pipe(delay(1000))))
  .subscribe(console.log);
```

(Outputs A, then B after 1s, then C after 1s)

C. Filtering Operators

5. `filter()` - Emits only values that satisfy a condition

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

of(1, 2, 3, 4, 5)
  .pipe(filter(value => value % 2 === 0))
  .subscribe(console.log);
```

Output:

```
2
4
```

6. `take()` - Takes only the first `n` values

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

of(10, 20, 30, 40, 50)
  .pipe(take(3))
  .subscribe(console.log);
```

Output:

```
10
20
30
```

D. Combination Operators

7. `merge()` - Merges multiple Observables

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

const obs1 = of(1, 2, 3);
const obs2 = of(4, 5, 6);

merge(obs1, obs2).subscribe(console.log);
```

Output:

```
1
2
3
4
5
6
```

8. `combineLatest()` - Combines the latest values from multiple Observables

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

const obs1 = of('Apple', 'Banana', 'Mango');
const obs2 = of('Red', 'Yellow', 'Green');

combineLatest([obs1, obs2]).subscribe(console.log);
```

Output:

```
 ['Mango', 'Green']
```

(The last emitted values are combined)

E. Error Handling Operators

9. `catchError()` - Catches errors and provides a fallback

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

throwError('Error Occurred!')
  .pipe(catchError(err => of('Fallback Value')))
  .subscribe(console.log);
```

Output:

```
Fallback Value
```

10. `retry()` - Retries a failed Observable

```
import { throwError } from 'rxjs';
import { retry, catchError } from 'rxjs/operators';

throwError('Network Error!')
  .pipe(
    retry(2), // Retry 2 times
    catchError(err => of('Final Fallback Value'))
  )
  .subscribe(console.log);
```

Output:

```
Final Fallback Value
```

3. Real-World Example (E-commerce Order Status)

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
import { map, filter } from 'rxjs/operators';

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

  orderStatus$ = new Observable<string>(observer => {
    observer.next('Order Placed');
    setTimeout(() => observer.next('Shipped'), 2000);
    setTimeout(() => observer.next('Out for Delivery'), 4000);
    setTimeout(() => observer.next('Delivered'), 6000);
    setTimeout(() => observer.complete(), 7000);
  });

  constructor() {
    this.orderStatus$
      .pipe(
        map(status => `Your Order Status: ${status}`),
        filter(status => !status.includes('Shipped')) // Filter out
'Shipped'
      )
      .subscribe({
        next: status => console.log(status),
        complete: () => console.log('Order Tracking Completed!')
      });
  }
}
```

Output (with delay):

```
Your Order Status: Order Placed
Your Order Status: Out for Delivery
Your Order Status: Delivered
Order Tracking Completed!
```

- The `filter()` operator removed `Shipped` from the output.
- The `map()` operator modified the emitted values.