RxJs

RxJS: Reactive Extension for JavaScript

# Introduction

**RxJS (Reactive Extensions for JavaScript)** is a library that allows you to **work with asynchronous data as streams**, using **Observables** to emit data over time and **operators** to transform, filter, and combine these streams efficiently.

Example:

"In Angular, HTTP requests using HttpClient return an Observable, which represents a stream of data over time. We can use RxJS operators like map to transform the response, for example extracting only the user names, and catchError to handle any errors gracefully. In the component, we subscribe to the Observable to actually execute the request and receive the data. This approach keeps the code declarative, reactive, and easy to manage for asynchronous operations."

# Stream:

"A stream is a sequence of data or events over time. In RxJS, we represent streams using Observables, which can emit multiple values asynchronously. You can listen to a stream, transform its data using operators like map or filter, and handle events as they occur, such as clicks, API responses, or timers."

# Difference between Array and Streams

| **Aspect** | **Array** | **Stream (Observable)** |
| --- | --- | --- |
| **Nature** | Collection of data **available all at once** | Sequence of data/events **over time** |
| **Execution** | **Eager** – data exists immediately | **Lazy** – emits data only when subscribed |
| **Size** | Fixed (or pre-defined) | Can be infinite (e.g., timer, clicks) |
| **Access** | Indexed access, synchronous | Event-driven, asynchronous |
| **Modification** | Can directly push/pop elements | Cannot modify directly; operators transform streams |
| **Use Case** | Static data | Dynamic or asynchronous data like clicks, HTTP requests, WebSocket messages |

"An array is a static collection of values available all at once, while a stream (Observable) represents a sequence of values that arrive over time. Arrays are synchronous, streams are asynchronous and lazy—they only produce values when you subscribe. Streams are ideal for events, HTTP responses, and timers, whereas arrays are for static data."

# Key elements: Observable, observer, subscription

Observable:

An **Observable** is a **stream of data or events** that can emit **multiple values over time**. Think of it as a blueprint for a sequence of asynchronous values.

import { of } from 'rxjs';

const numbers$ = of(1, 2, 3); // Observable emitting 3 numbers

Observer:

An **Observer** is an object that **reacts to values** emitted by an Observable. It defines **how to handle data, errors, and completion**.

Has three callbacks:

* next → handles emitted values
* error → handles errors
* complete → handles completion of the stream

const observer = {

next: (value: number) => console.log(value),

error: (err: any) => console.error(err),

complete: () => console.log('Stream completed')

};

Subscription:

A **Subscription** represents the **execution of an Observable**. It allows you to **start receiving values** and **unsubscribe** to stop listening.

Created when you call subscribe() on an Observable.

Provides a way to **clean up resources** and **prevent memory leaks**.

const numbers$ = of(1, 2, 3);

const subscription = numbers$.subscribe(observer);

// To stop receiving values

subscription.unsubscribe();

"In RxJS, an Observable defines a stream of data that can emit multiple values over time. An Observer is an object that reacts to the values, errors, and completion of that stream. When we subscribe to an Observable, it returns a Subscription, which allows us to start receiving data and also unsubscribe to clean up resources."

# Types of Observables:

“In RxJS, an **Observable** represents a stream of data that can emit values over time.  
There are different types of Observables based on how they emit data and how subscribers receive it.”

Cold Observables:

Cold Observables start emitting **only when a subscriber subscribes**.  
Each subscriber gets **its own independent stream** of data.

Example: of(), from(), interval(), http.get()

Code:

import { of } from 'rxjs';

const numbers$ = of(1, 2, 3);

numbers$.subscribe(val => console.log('Subscriber A:', val));

numbers$.subscribe(val => console.log('Subscriber B:', val));

Output: Both A and B get 1, 2, 3 separately (new execution each time).

Hot Observables:

Hot Observables **emit values whether you subscribe or not**.  
New subscribers may **miss previously emitted values**.

Example: Subject, fromEvent, WebSocket, interval() shared via share() operator

Code:

import { fromEvent } from 'rxjs';

const clicks$ = fromEvent(document, 'click');

clicks$.subscribe(() => console.log('Button clicked'));

Output: The click stream already exists — it doesn’t restart per subscriber.

Subjects (Special Type of Hot Observable):

Subjects act as **both an Observable and an Observer** — they **emit** and **listen** to data.

Subject:

Doesn’t store previous values.

All subscribers get **same data at the same time**.

Code:

const subject = new Subject<number>();

subject.subscribe(val => console.log('A:', val));

subject.next(1);

subject.subscribe(val => console.log('B:', val));

subject.next(2);

// Output: A:1, A:2, B:2

BehaviorSubject:

**Stores the latest value** and immediately emits it to new subscribers

Code:

const subject = new BehaviorSubject<number>(0);

subject.next(1);

subject.subscribe(val => console.log(val)); // gets 1 immediately

“Used when you always want to provide the most recent value, such as user login state.”

ReplaySubject:

**Replays previous N values** to new subscribers.

Code:

const subject = new ReplaySubject(2);

subject.next(1);

subject.next(2);

subject.next(3);

subject.subscribe(val => console.log(val)); // 2, 3

“Used when you want subscribers to get a history of values.”

AsyncSubject:

Emits **only the last value** when the stream **completes**.

Code:

const subject = new AsyncSubject();

subject.next(1);

subject.next(2);

subject.complete();

subject.subscribe(val => console.log(val)); // only 2

“Useful when you only need the final result, like an HTTP request.”

# Cold and Hot Observables:

**Cold Observables** produce values *per subscriber* (each subscription triggers a new execution).  
**Hot Observables** share the same execution for all subscribers (values are produced whether or not a particular subscriber is listening).

Cold Observables:

Cold: the producer is created inside the Observable and **starts when you subscribe**.

Each subscriber gets its **own independent** execution (own timer, own HTTP request, etc.).

Examples: of(), from(), new Observable(...), http.get() (Angular HttpClient returns a cold Observable).

Code:

import { Observable } from 'rxjs';

const cold$ = new Observable<number>((subscriber) => {

console.log('Observable starts (new execution)');

let i = 0;

const id = setInterval(() => {

i++;

subscriber.next(i);

if (i === 3) {

subscriber.complete();

clearInterval(id);

}

}, 1000);

// teardown logic

return () => {

console.log('Observable torn down');

clearInterval(id);

};

});

console.log('Subscribe A');

const subA = cold$.subscribe({

next: v => console.log('A next', v),

complete: () => console.log('A complete')

});

setTimeout(() => {

console.log('Subscribe B (after 1500ms)');

const subB = cold$.subscribe({

next: v => console.log('B next', v),

complete: () => console.log('B complete')

});

}, 1500);

OUTPUT:

Subscribe A

Observable starts (new execution)

A next 1

Subscribe B (after 1500ms)

Observable starts (new execution) <-- started again for B

A next 2

B next 1

A next 3

A complete

B next 2

B next 3

B complete

Emphasize *lazy* nature: nothing happens until subscribe() is called.

Useful when each subscriber should have fresh data/state (e.g., separate HTTP requests).

Hot Observables:

Hot: the producer exists **independently** of subscribers; emissions happen regardless of whether a subscriber is present.

Subscribers share the same source and may **miss** values emitted before subscribing.

Examples: fromEvent(document, 'click'), WebSocket streams, Subject and its variants, interval() turned hot via share()/Subject

Code:

import { interval, Subject } from 'rxjs';

import { take } from 'rxjs/operators';

// source$ is cold: each subscribe would start its own interval

const source$ = interval(1000).pipe(take(5));

// create a Subject (hot) and forward source emissions into it

const subject = new Subject<number>();

source$.subscribe(subject); // source now producing values once, feeding subject

console.log('Subscribe A to subject');

subject.subscribe(v => console.log('A', v));

setTimeout(() => {

console.log('Subscribe B to subject (after 2500ms)');

subject.subscribe(v => console.log('B', v));

}, 2500);

interval starts once (because we subscribed subject to it).

Subscriber A sees values from the start.

Subscriber B (later) misses the earlier emissions and only sees values emitted after it subscribes.

Hot Observables are good for shared resources (user events, websockets).

Be careful: late subscribers can miss values. Use BehaviorSubject / ReplaySubject to provide last values or a history.

# Creation Functions

**Creation functions** are **RxJS factory methods** that help you **create Observables easily** — instead of manually using new Observable().  
They cover most use cases like arrays, promises, intervals, timers, events, HTTP, etc.

| **Function** | **Description** | **Common Use Case** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **of()** | Emits a fixed set of values | Emit static values, quick demos |

|  |  |  |
| --- | --- | --- |
| **from()** | Converts arrays, promises, or iterables to observables | Convert existing data structures |

|  |  |  |
| --- | --- | --- |
| **interval()** | Emits sequential numbers at regular intervals | Timers, counters, periodic tasks |

|  |  |  |
| --- | --- | --- |
| **timer()** | Emits once after a delay, or repeatedly after delay + interval | Delay tasks, countdowns |

|  |  |  |
| --- | --- | --- |
| **fromEvent()** | Converts DOM events to observables | Event handling (clicks, scrolls) |

|  |  |  |
| --- | --- | --- |
| **range()** | Emits numbers in a specified range | Generating numeric sequences |

|  |  |  |
| --- | --- | --- |
| **throwError()** | Creates observable that immediately errors | Testing, error handling |

|  |  |  |
| --- | --- | --- |
| **empty() / EMPTY** | Creates observable that completes immediately | No emission, just completion |

|  |  |  |
| --- | --- | --- |
| **defer()** | Waits until subscription to create Observable | Lazy creation (useful for conditional or dynamic sources) |

|  |  |  |
| --- | --- | --- |
| **generate()** | Emits values using a loop-like logic | Advanced dynamic sequences |

|  |  |  |
| --- | --- | --- |
| **ajax()** | Creates observable for HTTP requests | Direct HTTP (outside Angular HttpClient) |
|  |  |  |

Of() – emit static values

“of() emits the provided values synchronously and then completes. It’s useful for testing or providing static data.”

Code:

import { of } from 'rxjs';

const source$ = of(10, 20, 30);

source$.subscribe({

next: val => console.log('Value:', val),

complete: () => console.log('Complete')

});

Output:

Value: 10

Value: 20

Value: 30

Complete

From() - Convert iterable/promise/array

“from() converts arrays, promises, or iterable data into an Observable. For example, Angular’s HttpClient internally uses something like from(fetch(...)) under the hood.”

Code:

import { from } from 'rxjs';

// from Array

from([1, 2, 3]).subscribe(v => console.log('Array:', v));

// from Promise

from(fetch('https://jsonplaceholder.typicode.com/posts/1'))

.subscribe(res => console.log('Promise:', res));

from() — convert iterable/promise/array

“from() converts arrays, promises, or iterable data into an Observable. For example, Angular’s HttpClient internally uses something like from(fetch(...)) under the hood.”

CODE:

import { from } from 'rxjs';

// from Array

from([1, 2, 3]).subscribe(v => console.log('Array:', v));

// from Promise

from(fetch('https://jsonplaceholder.typicode.com/posts/1'))

.subscribe(res => console.log('Promise:', res));

timer() — delay + interval

“timer() is like an enhanced interval() — it can delay the first emission or emit just once.”

CODE:

import { timer } from 'rxjs';

// emit first after 3s, then every 1s

const source$ = timer(3000, 1000);

source$.subscribe(val => console.log('Timer:', val));

fromEvent() — convert event to observable

“fromEvent() turns DOM events into a stream — perfect for reactive UI logic in Angular.”

CODE:

import { fromEvent } from 'rxjs';

const clicks$ = fromEvent(document, 'click');

clicks$.subscribe(event => console.log('Clicked:', event));

range() — numeric sequence

“range() emits a sequence of numbers starting from a given value. It’s synchronous and completes after emission.”

CODE:

import { range } from 'rxjs';

range(1, 5).subscribe(val => console.log('Range:', val));

OUTPUT:

Range: 1

Range: 2

Range: 3

Range: 4

Range: 5

throwError() — emit an error immediately

“throwError() is used for testing or creating error streams in RxJS pipelines.”

CODE:

import { throwError } from 'rxjs';

const error$ = throwError(() => new Error('Something went wrong!'));

error$.subscribe({

next: val => console.log(val),

error: err => console.error('Error:', err.message)

});

EMPTY — emits nothing, just completes

“EMPTY emits nothing but completes instantly. Often used in conditional logic with switchMap when no data needs to be emitted.”

CODE:

import { EMPTY } from 'rxjs';

EMPTY.subscribe({

next: v => console.log(v),

complete: () => console.log('Completed immediately')

});

# ForkJoin - Wait for all Observables to complete

"forkJoin in RxJS is used when you want to make multiple Observables run in parallel, and you need the final result only after *all of them complete*. It emits a single combined value — the last emitted value from each observable — and then completes."

CODE:

const user1$ = this.http.get('https://jsonplaceholder.typicode.com/users/1');

const user2$ = this.http.get('https://jsonplaceholder.typicode.com/users/2');

const user3$ = this.http.get('https://jsonplaceholder.typicode.com/users/3');

forkJoin([user1$, user2$, user3$]).subscribe(([user1, user2, user3]) => {

console.log('User 1:', user1.username);

console.log('User 2:', user2.username);

console.log('User 3:', user3.username);

});

“In this example, all three HTTP requests are made in parallel.  
forkJoin waits until all three complete and then gives a single array of results.  
This is useful for scenarios like loading a dashboard where all data must be fetched before displaying the page.”

# combineLatest() - Emit whenever any source emits (after all have emitted once)

"combineLatest emits a new combined value whenever any of the source Observables emit, but only *after all Observables have emitted at least once.*"

CODE:

import { combineLatest, interval } from 'rxjs';

import { map, take } from 'rxjs/operators';

// Simulate two Observables

const timer1$ = interval(1000).pipe(take(3)); // emits 0,1,2

const timer2$ = interval(2000).pipe(take(2)); // emits 0,1

combineLatest([timer1$, timer2$]).subscribe(([val1, val2]) => {

console.log(`Timer1: ${val1}, Timer2: ${val2}`);

});

“combineLatest will start emitting only after both Observables emit at least once.  
After that, whenever any Observable emits a new value, it emits a new combined array.  
It’s ideal when you want live updates based on multiple changing sources.”

# Pipeable Operators