**Angular**

**1. What is RxJS and why is it used in Angular?**

**Answer:**  
RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using Observables. Angular uses RxJS to handle asynchronous operations such as HTTP requests, user input events, and more.

**Why RxJS in Angular?**

* Composable: Chain multiple operations.
* Declarative: Define the flow of data.
* Async handling: Ideal for HTTP calls, event streams, etc.

**2. Difference between Observable and Promise?**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Observable** | **Promise** |
| Eager/Lazy | Lazy (executes on subscribe) | Eager |
| Multiple Values | Yes (stream) | No (single value) |
| Operators | Rich set of operators | No operators |
| Cancelable | Yes | No |

**Example:**

**// Observable**

const obs$ = new Observable(observer => {

observer.next('Hello');

observer.complete();

});

obs$.subscribe(val => console.log(val));

**// Promise**

const promise = new Promise(resolve => resolve('Hello'));

promise.then(val => console.log(val));

**3. What are the different types of subjects in RxJS?**

1. **Subject:** Multicast observable.
2. **BehaviorSubject:** Holds and emits the last value.
3. **ReplaySubject:** Replays a specified number of previous values.
4. **AsyncSubject:** Emits last value on complete only.

**Example:**

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

subject.subscribe(val => console.log('Sub 1:', val));

subject.next(1);

subject.subscribe(val => console.log('Sub 2:', val)); // Gets 1

**4. What are some commonly used RxJS operators?**

* **Creation operators:** of, from, interval, timer
* **Transformation:** map, switchMap, mergeMap, concatMap
* **Filtering:** filter, debounceTime, distinctUntilChanged
* **Combination:** merge, concat, combineLatest, forkJoin
* **Utility:** tap, take, retry, catchError

**5. What is the difference between switchMap, mergeMap, concatMap, and exhaustMap?**

|  |  |
| --- | --- |
| **Operator** | **Behavior** |
| switchMap | Cancels previous inner observable, switches to new one |
| mergeMap | Subscribes to all inner observables simultaneously |
| concatMap | Queues inner observables and subscribes one after the other |
| exhaustMap | Ignores new inner observables while a previous one is still active |

**Example with switchMap:**

this.searchInput.valueChanges

.pipe(

debounceTime(300),

switchMap(value => this.http.get(`/api/search?q=${value}`))

)

.subscribe(data => this.results = data);

**6. How do you handle errors in RxJS streams?**

Use catchError() and optionally retry():

this.http.get('/api/data')

.pipe(

retry(2),

catchError(error => {

console.error(error);

return of([]); // fallback value

})

)

.subscribe();

**7. What is the difference between cold and hot Observables?**

* **Cold Observable:** Starts emitting values only when subscribed (e.g., http.get()).
* **Hot Observable:** Already producing values even before subscription (e.g., Subject, DOM events).

**8. How do you unsubscribe from Observables in Angular?**

* **Manually:** Store subscription and call .unsubscribe()
* **With takeUntil:** Use a Subject to complete stream
* **Async Pipe:** Handles subscription and unsubscription automatically in templates

ngOnDestroy() {

this.destroy$.next(true);

this.destroy$.complete();

}

**9. What is debounceTime and when do you use it?**

Delays emission of values from source observable by the given time.

**Use Case:** Debounce user input (e.g., search box)

this.searchInput.valueChanges

.pipe(debounceTime(300))

.subscribe(value => this.search(value));

**10. What is the use of combineLatest vs forkJoin?**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Emits When** | **Use Case** |
| combineLatest | Any source emits | Combine live values from streams |
| forkJoin | All sources complete | Combine final values from multiple requests |

**Creation operators in angular with example**

### 1. of : Creates an observable from static values.

import { of } from 'rxjs';

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

obs$.subscribe(value => console.log(value));

// Output:

// 1

// 2

// 3

## **of() in Angular – Mock API Response in a Service**

// data.service.ts

import { Injectable } from '@angular/core';

import { of } from 'rxjs';

@Injectable({ providedIn: 'root' })

export class DataService {

getStaticData() {

return of(['Angular', 'React', 'Vue']);

}

}

// component.ts

this.dataService.getStaticData().subscribe(data => {

console.log('Frameworks:', data);

});

**2. from : Creates an observable from an array, promise, or iterable.**

import { from } from 'rxjs';

const obsFromArray$ = from([10, 20, 30]);

obsFromArray$.subscribe(value => console.log(value));

// Output:

// 10

// 20

// 30

const promise = Promise.resolve('Resolved!');

const obsFromPromise$ = from(promise);

obsFromPromise$.subscribe(value => console.log(value));

// Output:

// Resolved!

## **from() –** **Convert a Promise to Observable**

// data.service.ts

import { from } from 'rxjs';

getPromiseData() {

const promise = fetch('https://jsonplaceholder.typicode.com/posts/1').then(res => res.json());

return from(promise); // Convert promise to observable

}

// component.ts

this.dataService.getPromiseData().subscribe(data => {

console.log('Post:', data);

});

### 3. interval : Emits a sequence of numbers every specified interval of time (in ms).

import { interval } from 'rxjs';

import { take } from 'rxjs/operators';

const obs$ = interval(1000).pipe(take(3));

obs$.subscribe(value => console.log(value));

// Output (every 1 second):

// 0

// 1

// 2

## **interval() –** **Auto-refresh data every few seconds**

import { interval, switchMap } from 'rxjs';

ngOnInit() {

interval(5000).pipe(

switchMap(() => this.dataService.getLiveData())

).subscribe(data => {

this.liveData = data;

});

}

⚠️ getLiveData() should return an Observable (e.g., HttpClient.get()).

**4. timer : Emits a number after a delay, and optionally continues to emit sequential numbers.**

import { timer } from 'rxjs';

// Emit once after 2 seconds

const obs$ = timer(2000);

obs$.subscribe(() => console.log('Timer done'));

// Emit every 1s, starting after 3s

const repeated$ = timer(3000, 1000);

repeated$.subscribe(value => console.log(value)); // 0, 1, 2, ...

## **timer() – Show a message after delay**

**ts**

import { timer } from 'rxjs';

ngOnInit() {

timer(3000).subscribe(() => {

this.showMessage = true;

});

}

**html**

<div \*ngIf="showMessage">This appeared after 3 seconds!</div>

### 5. range : Emits a range of sequential numbers.

ts

import { range } from 'rxjs';

const obs$ = range(5, 3); // Starts from 5, emits 3 values

obs$.subscribe(value => console.log(value));

// Output:

// 5

// 6

// 7

## range() – **Render a range in UI**

ts

import { range } from 'rxjs';

ngOnInit() {

range(1, 5).subscribe(num => {

this.rangeList.push(num);

});

}

**html**

<ul>

<li \*ngFor="let num of rangeList">{{ num }}</li>

</ul>

### 6. fromEvent : Creates an observable from DOM events (commonly used in Angular).

ts

import { fromEvent } from 'rxjs';

// Inside a component

ngOnInit() {

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

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

}

## fromEvent() – **React to DOM Events**

ts

import { fromEvent } from 'rxjs';

import { throttleTime } from 'rxjs/operators';

ngOnInit() {

const click$ = fromEvent(document, 'click').pipe(

throttleTime(1000)

);

click$.subscribe(event => {

console.log('Clicked at:', event.timeStamp);

});

}

**IQ: What is transformation in rxjs with error handling and filtering operators real-time Angular example?**

**Answer:**

In **RxJS**, **transformation operators** are used to **modify or transform the data** emitted by Observables. **Error handling** and **filtering operators** help you manage bad data or exceptions and only process the relevant data.

Let’s walk through a **real-time Angular example** that includes:

* 🔄 **Transformation:** using map, switchMap
* 🧹 **Filtering:** using filter
* ⚠️ **Error Handling:** using catchError

## 🔧 Real-time Use Case: **Search with Input, Filter by Length, Transform Data, Handle API Errors**

### 🧪 Scenario:

We want to implement a **live search** feature:

* Listen to user input.
* Ignore input shorter than 3 characters.
* Send HTTP request using switchMap.
* Transform response using map.
* Catch errors using catchError.

### 📁 search.service.ts

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable, of } from 'rxjs';

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

@Injectable({ providedIn: 'root' })

export class SearchService {

constructor(private http: HttpClient) {}

search(term: string): Observable<any[]> {

return this.http.get<any[]>(`https://api.example.com/search?q=${term}`).pipe(

map(results => results.slice(0, 5)), // transformation: return top 5 results

catchError(error => {

console.error('Search failed', error);

return of([]); // return empty array on error

})

);

}

}

### 📁 search.component.ts

import { Component, OnInit } from '@angular/core';

import { FormControl } from '@angular/forms';

import { debounceTime, filter, switchMap } from 'rxjs/operators';

import { SearchService } from './search.service';

@Component({

selector: 'app-search',

template: `

<input type="text" [formControl]="searchControl" placeholder="Search..." />

<ul>

<li \*ngFor="let item of results">{{ item.name }}</li>

</ul>

`

})

export class SearchComponent implements OnInit {

searchControl = new FormControl('');

results: any[] = [];

constructor(private searchService: SearchService) {}

ngOnInit() {

this.searchControl.valueChanges.pipe(

debounceTime(300), // Wait for 300ms pause

filter(term => term.length > 2), // Filter short inputs

switchMap(term => this.searchService.search(term)) // Transformation and error handling in service

).subscribe(data => {

this.results = data;

});

}

}

### ✅ RxJS Operators Breakdown:

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Purpose** |
| Transformation | map | Modify response data (e.g. take top 5 results) |
|  | switchMap | Cancel previous request if a new value comes |
| Filtering | filter | Only allow terms > 2 chars |
|  | debounceTime | Reduce noise from fast typing |
| Error Handling | catchError | Return fallback data and log error |

### ****RxJS Operators Cheat Sheet****

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Operator** | **Use Case** | **Example** |
| **Transformation** | map() | Transforms each emitted value. | obs$.pipe(map(val => val \* 2)) |
|  | switchMap() | Cancels previous HTTP request and switches to a new observable. | obs$.pipe(switchMap(val => http.get(val))) |
|  | mergeMap() | Maps to multiple observables and merges the results. | obs$.pipe(mergeMap(val => http.get(val))) |
|  | concatMap() | Maps to observables and executes them in order. | obs$.pipe(concatMap(val => http.get(val))) |
| **Filtering** | filter() | Filters values based on a condition. | obs$.pipe(filter(val => val > 10)) |
|  | debounceTime() | Emits values after a delay, ignoring rapid emissions. | obs$.pipe(debounceTime(300)) |
|  | distinctUntilChanged() | Only emits values that are different from the previous one. | obs$.pipe(distinctUntilChanged()) |
| **Error Handling** | catchError() | Catches errors and returns a fallback observable. | obs$.pipe(catchError(err => of([]))) |
|  | retry() | Retries the observable in case of an error. | obs$.pipe(retry(3)) |
|  | retryWhen() | Retries based on custom logic or condition. | obs$.pipe(retryWhen(errors => errors.delay(1000))) |
| **Combination** | combineLatest() | Combines the latest values from multiple observables. | combineLatest(obs1$, obs2$) |
|  | forkJoin() | Waits for all observables to complete and then emits the last value from each. | forkJoin([obs1$, obs2$]) |
|  | zip() | Combines observables in pairs, emitting the values together. | zip(obs1$, obs2$) |
| **Utility** | tap() | Performs side-effects for debugging without changing the stream. | obs$.pipe(tap(val => console.log(val))) |
|  | finalize() | Executes logic when the observable completes or errors. | obs$.pipe(finalize(() => console.log('Completed!'))) |

### ****Examples****

1. **map()** — Transforms the emitted value.

import { of } from 'rxjs';

import { map } from 'rxjs/operators';

of(1, 2, 3).pipe(

map(val => val \* 2)

).subscribe(result => console.log(result));

// Output: 2, 4, 6

1. **switchMap()** — Cancels the previous observable and subscribes to a new one.

import { of } from 'rxjs';

import { switchMap } from 'rxjs/operators';

of('https://jsonplaceholder.typicode.com/posts').pipe(

switchMap(url => fetch(url).then(res => res.json()))

).subscribe(data => console.log(data));

1. **debounceTime()** — Waits for a pause in emissions before emitting.

import { of } from 'rxjs';

import { debounceTime } from 'rxjs/operators';

of('a', 'b', 'c').pipe(

debounceTime(500)

).subscribe(val => console.log(val));

// It waits for 500ms of no emissions before emitting the value.

1. **catchError()** — Catches errors and returns a fallback observable.

import { of } from 'rxjs';

import { catchError } from 'rxjs/operators';

of(1, 2, 3).pipe(

catchError(err => of('An error occurred!'))

).subscribe(result => console.log(result));

// Output: 1, 2, 3

1. **retry()** — Retries the observable a specified number of times if it errors.

import { of } from 'rxjs';

import { retry } from 'rxjs/operators';

let count = 0;

of(1).pipe(

retry(3),

map(() => {

if (count++ < 2) {

throw new Error('Error!');

}

return 'Success';

})

).subscribe(

data => console.log(data),

err => console.log(err)

);

// Will retry 3 times before logging the error

1. **combineLatest()** — Combines the latest values from multiple observables.

import { of, combineLatest } from 'rxjs';

const obs1$ = of('A', 'B');

const obs2$ = of(1, 2);

combineLatest([obs1$, obs2$]).subscribe(([val1, val2]) => {

console.log(val1, val2);

});

// Output: 'B' 2 (latest values)

**WEB API**

**SQL Server**