**Chapter 11: RxJS & Observables**

**What is RxJS?**

**RxJS (Reactive Extensions for JavaScript)** is a library for working with asynchronous data streams using **Observables**.

It’s used extensively in Angular for:

* Handling HTTP requests
* Listening to user input
* Managing state

**Importing from RxJS**

Common imports from RxJS:

import { Observable, of, from, interval } from 'rxjs';

**Observable – The Core of RxJS**

An **Observable** is a way to **handle asynchronous data streams** in Angular. It's like a pipe that data flows through over time. You can subscribe to it and react when data is emitted.

**Key Terms:**

* **Observable**: The data source (stream).
* **Observer**: The consumer that listens to the data stream.
* **Subscription**: Connection to the observable to start receiving data.

### Why use it?

In Angular, HTTP requests, user events, and forms emit data **over time**. Instead of writing a lot of callback or Promise code, you use Observables to reactively manage streams of data.

**Observable Examples**

**1. Simple Observable for Emitting Static Values**

**✅ Scenario: Emitting a fixed set of values one after another**

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

import { Observable } from 'rxjs';

@Component({

selector: 'app-basic-observable',

template: `<p>Check the console for emitted values.</p>`

})

export class BasicObservableComponent implements OnInit {

ngOnInit() {

const numberStream = new Observable<number>(observer => {

observer.next(1);

observer.next(2);

observer.next(3);

observer.complete();

});

numberStream.subscribe({

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

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

});

}

}

**Explanation:**

* Emits static values 1, 2, 3.
* next() sends the value.
* complete() ends the stream.

**2. Observable with Timed Emissions using interval()**

**✅ Scenario: Auto-incrementing counter every second**

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

import { interval } from 'rxjs';

@Component({

selector: 'app-interval-example',

template: `<p>Counter: {{ counter }}</p>`

})

export class IntervalExampleComponent implements OnInit {

counter = 0;

ngOnInit() {

interval(1000).subscribe(val => {

this.counter = val;

});

}

}

**Explanation:**

* interval(1000) emits values starting from 0 every second.
* Used in timers, real-time dashboards, etc.

**3. Subscription with next, error & complete**

When you subscribe to an observable, you can pass an object with up to **three callbacks**:

observable.subscribe({

next: (value) => { /\* called on every emitted value \*/ },

error: (err) => { /\* called if an error occurs, stream terminated \*/ },

complete: () => { /\* called when the observable completes \*/ }

});

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

import { Observable } from 'rxjs';

@Component({

selector: 'app-subscription-example',

template: `<p>Check the console output</p>`

})

export class SubscriptionExampleComponent implements OnInit {

ngOnInit() {

const myObservable = new Observable<number>(observer => {

observer.next(1);

observer.next(2);

observer.error('Something went wrong!');

observer.next(3); // Will NOT be called

observer.complete(); // Will NOT be called

});

myObservable.subscribe({

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

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

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

});

}

}

**Output:**

Value: 1

Value: 2

Error: Something went wrong!

* next(3) and complete() are **not called** after error().

The .subscribe() method can be used in **two different styles**:

**✅ 1. Object Syntax (Named Handlers)**

This is the **explicit way** to handle all possible observable events using named keys: next, error, and complete.

**Syntax:**

observable.subscribe({

next: (value) => { ... },

error: (err) => { ... },

complete: () => { ... }

});

**Example:**

import { of } from 'rxjs';

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

numbers$.subscribe({

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

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

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

});

**✅ Best for:**

* Readability
* Using multiple callbacks
* Larger codebases or teams

**✅ 2. Positional Syntax (Without Keys)**

This is the **shorter, functional way** to subscribe using arguments in order:

**Syntax:**

observable.subscribe(

nextFn,

errorFn, // optional

completeFn // optional

);

**Example:**

import { of } from 'rxjs';

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

numbers$.subscribe(

value => console.log('Received:', value), // next

error => console.error('Error:', error), // error

() => console.log('Stream complete!') // complete

);

**Subscribe with next() only**

If you only need the emitted values:

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

**Subscribe with next() and complete(), but no error()**

You **must pass null** as the second argument:

numbers$.subscribe(

value => console.log(value),

null,

() => console.log('All done!')

);

**4. Observable for HTTP Request (API Integration)**

**✅ Scenario: Fetch user data from a REST API**

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

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

@Component({

selector: 'app-api-user-list',

template: `

<h3>User List</h3>

<ul>

<li \*ngFor="let user of users">{{ user.name }}</li>

</ul>

`

})

export class ApiUserListComponent implements OnInit {

users: any[] = [];

constructor(private http: HttpClient) {}

ngOnInit() {

this.http.get<any[]>('https://jsonplaceholder.typicode.com/users')

.subscribe(data => {

this.users = data;

});

}

}

**Explanation:**

* HttpClient.get() returns an Observable.
* Data is fetched asynchronously and displayed in the template.

**RxJS Operators**

**RxJS operators** are pure functions used with .pipe() to **transform, filter, combine, and manage** Observable streams. They turn basic Observables into powerful, composable data flows. Operators are categorized into:

1. **Creation** – for producing Observables
2. **Transformation** – to map or flatten values
3. **Filtering** – to trim or refine emitted values
4. **Utility** – for side effects, error handling, sharing, etc.

### ✅ ****Operators****

| **Operator** | **Usage** |
| --- | --- |
| of() | Emits given values sequentially and completes. |
| from() | Converts an array, promise, or iterable into an observable. |
| interval() | Emits an increasing number at a specified time interval. |
| timer() | Emits once after delay or periodically after initial delay. |
| forkJoin() | Waits for multiple observables to complete, then emits their last values as an array. |
| map() | Transforms each emitted value using a provided function. |
| switchMap() | Cancels previous inner observable and switches to a new one on each emission. |
| mergeMap() | Subscribes to multiple inner observables simultaneously. |
| concatMap() | Queues inner observables and processes them one after another. |
| filter() | Emits only values that satisfy a condition. |
| debounceTime() | Emits a value only after a silence period. |
| distinctUntilChanged() | Emits a value only if it’s different from the previous one. |
| take() | Emits only the first n values then completes. |
| first() | Emits the first value that meets a condition (or first at all). |
| tap() | Executes side-effects (like logging) without modifying values. |
| catchError() | Handles errors and returns a fallback observable. |
| retry() | Automatically resubscribes on error, up to a given count. |
| share() | Shares a single subscription among multiple subscribers. |

**Program 1:**

import { of } from 'rxjs';

const numbers$ = of(1, 2, 3, 4, 5);

numbers$.subscribe({

next: value => console.log('Received:', value),

complete: () => console.log('Done emitting!')

});

**Program 2:**

import { of } from 'rxjs';

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

// Create observable from numbers 1 to 5

const numbers$ = of(1, 2, 3, 4, 5).pipe(

map(num => num \* 10), // Multiply each number by 10

filter(num => num > 20) // Only allow numbers > 20

);

// Subscribe to the observable

numbers$.subscribe({

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

complete: () => console.log('Done!')

});

**Expected Console Output**

Output: 30

Output: 40

Output: 50

Done!

**Program 3:**

import { interval } from 'rxjs';

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

// Emits numbers: 0, 1, 2, 3, 4, 5, ...

const source$ = interval(1000).pipe(

filter(num => num % 2 === 0), // Keep only even numbers: 0, 2...

take(3) // Take only the first 3 even numbers

);

source$.subscribe({

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

complete: () => console.log('Completed after 3 even numbers!')

});

**Expected Console Output (1 sec apart)**

Received even: 0

Received even: 2

Received even: 4

Completed after 3 even numbers!

**Program 4:**

**✅ Angular Component Code**

// user-list.component.ts

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

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

import { of } from 'rxjs';

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

@Component({

selector: 'app-user-list',

templateUrl: './user-list.component.html'

})

export class UserListComponent implements OnInit {

users: any[] = [];

usersUrl = 'https://jsonplaceholder.typicode.com/users';

constructor(private http: HttpClient) {}

ngOnInit(): void {

this.fetchUsers();

}

fetchUsers(): void {

this.http.get<any[]>(this.usersUrl).pipe(

// ✅ Side-effect for debugging

tap(data => console.log('Data fetched:', data)),

catchError(error => {

console.error('Error occurred', error);

return of([]);

})

).subscribe(data => {

this.users = data;

});

}

}

**HTML Template**

<!-- user-list.component.html -->

<h3>User List</h3>

<ul>

<li \*ngFor="let user of users">

{{ user.name }} ({{ user.email }})

</li>

</ul>

**Console Output (Success)**

Data fetched: [ { id: 1, name: 'Leanne Graham', email: 'leanne@example.com' }, ... ]

**Console Output (Error)**

Error occurred: HttpErrorResponse { ... }

**✅ Subject – A Multicasting Observable**

**Subject:**

A Subject is both an **Observable** and an **Observer**. It can:

* **Emit values** (next())
* Be **subscribed to** like any other Observable

**Use Case:**

Used when **multiple parts of your app need to share the same data stream** or when you want to **manually control emissions** (like emitting events).

**Example:**

import { Subject } from 'rxjs';

const subject = new Subject<number>();

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

subject.next(1);

subject.next(2);

subject.subscribe(val => console.log('Subscriber 2:', val));

subject.next(3);

**Output:**

Subscriber 1: 1

Subscriber 1: 2

Subscriber 1: 3

Subscriber 2: 3

**Note**: Subscriber 2 **only receives values emitted after** it subscribes.

**✅ BehaviorSubject – with Initial Value**

A BehaviorSubject is a type of Subject that **always holds the latest value** and **immediately emits it to new subscribers**.

**Use Case:**

Used for **state management** or when you want all subscribers to get the **current value immediately**, not just future emissions.

**Example:**

import { BehaviorSubject } from 'rxjs';

const behaviorSubject = new BehaviorSubject<number>(0); // initial value = 0

behaviorSubject.subscribe(val => console.log('Subscriber 1:', val));

behaviorSubject.next(1);

behaviorSubject.next(2);

behaviorSubject.subscribe(val => console.log('Subscriber 2:', val));

behaviorSubject.next(3);

**Output:**

Subscriber 1: 0

Subscriber 1: 1

Subscriber 1: 2

Subscriber 1: 3

Subscriber 2: 2

Subscriber 2: 3

**Note**: Subscriber 2 **immediately receives the current value (2)** when it subscribes.

**✅ AsyncPipe**

The AsyncPipe is a **built-in Angular pipe** that:

* **Subscribes** to an Observable or Promise
* **Unsubscribes automatically** when the component is destroyed
* **Pushes the emitted values** directly to the view

**Why Use AsyncPipe?**

Without AsyncPipe, you'd have to:

* Manually subscribe() in your component
* Store the emitted data in a variable
* Worry about unsubscribing to prevent memory leaks

With AsyncPipe, you do **all of that in the template.**

**Displaying Data from HTTP (AsyncPipe)**

**✅ Component (TypeScript)**

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

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

import { Observable } from 'rxjs';

@Component({

selector: 'app-user-list',

templateUrl: './user-list.component.html'

})

export class UserListComponent {

users$: Observable<any[]>;

constructor(private http: HttpClient) {

this.users$ = this.http.get<any[]>('https://jsonplaceholder.typicode.com/users');

}

}

**✅ Template (HTML)**

<ul>

<li \*ngFor="let user of users$ | async">

{{ user.name }} ({{ user.email }})

</li>

</ul>

✅ No need to manually subscribe() or handle cleanup.

✅ Handles API call, subscription, and updates the UI — all in the template.

**✅ Interceptor**

An **HTTP Interceptor** in Angular is a special service that **intercepts every HTTP request and/or response**, allowing you to:

**Why Use Interceptors?**

| **Use Case** | **Example** |
| --- | --- |
| Add auth token | Attach JWT token to every request |
| Log API traffic | Console.log all outgoing requests |
| Global error handling | Catch and handle 401, 500 errors |
| UI loader control | Show spinner on request start/end |
| Retry logic or offline caching | Retry failed calls automatically |

**1. Generate the Interceptor**

ng generate interceptor auth

This creates auth.interceptor.ts.

**2. Create the interceptor class**

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

import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class AuthInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const token = localStorage.getItem('authToken');

const authReq = token

? req.clone({

setHeaders: { Authorization: `Bearer ${token}` }

})

: req;

return next.handle(authReq);

}

}

**Explanation: const authReq = token ? ... : req;**

* This is a conditional (ternary) operator checking if the variable token exists (is truthy).
* If token **exists**, it:
  + Calls req.clone() to create a **new copy** of the original HTTP request.
  + Adds or updates the request headers with Authorization: Bearer <token>.
  + This is important because Angular's HttpRequest objects are **immutable**—you cannot change the original request directly, so you must clone it to modify headers.
* If token **does not exist**, it simply uses the original req without modification.

**Explanation: return next.handle(authReq);**

* Passes the (possibly modified) request authReq to the **next interceptor** in the chain or to the backend if this is the last interceptor.
* next.handle() returns an Observable<HttpEvent<any>> that Angular uses to continue processing the HTTP request and eventually send it to the server.

**In short:**

* **If there’s a token, add it as an Authorization header** using a cloned request.
* **Otherwise, just pass the original request unchanged.**
* Then forward that request to the next step in the HTTP pipeline.

**3. Provide the interceptor in your standalone root component (config.ts)**

import { bootstrapApplication } from '@angular/platform-browser';

import { provideHttpClient, withInterceptors } from '@angular/common/http';

import { AppComponent } from './app.component';

import { AuthInterceptor } from './auth.interceptor';

bootstrapApplication(AppComponent, {

providers: [

provideHttpClient(withInterceptors([AuthInterceptor]))

]

});

**Explanation:**

* provideHttpClient sets up Angular's HTTP client.
* withInterceptors([AuthInterceptor]) attaches your interceptor(s).
* This replaces the providers array in traditional NgModules.

**If you want to add multiple interceptors:**

provideHttpClient(withInterceptors([Interceptor1, Interceptor2]))