**Chapter 11: RxJS & Observables in Angular**

**🔍 What is RxJS?**

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

Angular uses RxJS **under the hood** in:

* HttpClient
* Reactive Forms
* Router events
* Signals (in newer versions)
* User input handling, polling, websockets, etc.

**🧠 Key RxJS Concepts**

| **Term** | **What It Means** |
| --- | --- |
| Observable | A stream of data you can subscribe to |
| Observer | A handler that reacts to values from observable |
| Operators | Functions to transform or filter the data stream |
| Subject | A multicast observable (can emit + listen) |

**🧱 Basic Observable Example**

**✅ Creating an Observable**

import { Observable } from 'rxjs';

const observable = new Observable<number>(subscriber => {

subscriber.next(1);

subscriber.next(2);

subscriber.complete();

});

**✅ Subscribing**

observable.subscribe({

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

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

});

**🌊 Observables with HttpClient**

Most real-world use comes from services like this:

getProducts(): Observable<Product[]> {

return this.http.get<Product[]>(this.apiUrl);

}

Then you subscribe in your component:

this.productService.getProducts().subscribe(products => {

this.products = products;

});

**⚙️ Most Common RxJS Operators**

| **Operator** | **Description** |
| --- | --- |
| map() | Transform each emitted value |
| filter() | Emit only values that match a condition |
| tap() | Perform side effects (e.g., logging) |
| switchMap() | Cancel previous observable and switch to a new one |
| catchError() | Handle errors and return fallback or rethrow |
| combineLatest() | Combine multiple observables |

**📦 Practical Example: switchMap for Live Search**

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

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

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

import { ProductService } from './services/product.service';

@Component({

standalone: true,

selector: 'app-product-search',

template: `

<input [formControl]="searchControl" placeholder="Search products" />

<ul>

<li \*ngFor="let product of products">{{ product.name }}</li>

</ul>

`,

imports: [ReactiveFormsModule]

})

export class ProductSearchComponent {

searchControl = new FormControl('');

products: any[] = [];

constructor(private productService: ProductService) {

this.searchControl.valueChanges

.pipe(

debounceTime(300),

switchMap(query => this.productService.searchProducts(query))

)

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

}

}

**🔄 Subjects & BehaviorSubjects**

**✅ Subject**

Can manually emit values and subscribe to them.

const subject = new Subject<number>();

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

subject.next(42);

**✅ BehaviorSubject**

Same as Subject but keeps the **last value** and emits it immediately to new subscribers.

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

counter.next(1); // All subscribers get 1

✅ Use in services to **share data across components**.

**🧪 Real-World: Sharing Data with BehaviorSubject**

**theme.service.ts**

@Injectable({ providedIn: 'root' })

export class ThemeService {

private theme$ = new BehaviorSubject<'light' | 'dark'>('light');

setTheme(theme: 'light' | 'dark') {

this.theme$.next(theme);

}

getTheme(): Observable<'light' | 'dark'> {

return this.theme$.asObservable();

}

}

**navbar.component.ts**

constructor(private themeService: ThemeService) {

this.themeService.getTheme().subscribe(mode => {

this.isDarkMode = mode === 'dark';

});

}

**🧠 Summary**

| **Feature** | **Use Case** |
| --- | --- |
| Observable | Streaming data (API, UI, events) |
| switchMap | Search, cancel old requests |
| catchError | Handle HTTP/API errors |
| Subject | Manual event stream (like emitters) |
| BehaviorSubject | Data sharing between components |

**✅ Exercises**

1. Build a search box using debounceTime() and switchMap() to query a fake API.
2. Create a service with a BehaviorSubject to store a global setting (e.g., app language).
3. Fetch data from API and transform with map() before displaying.
4. Handle API errors using catchError() and show a message.

To collaborate on a full document with detailed explanations, code breakdowns, and real-world use cases, you’ll need to be **logged in** to use **ChatGPT canvas**.

But no worries — I’ll continue right here by giving you a more **detailed explanation** of each concept, including **why, where, and when to use it**, and **line-by-line code explanations**.

# 📘 Deep Dive into RxJS for Angular Beginners

## ✅ 1. Observable – The Core of RxJS

### 🔍 What is it?

An **Observable** is like a function that emits multiple values over time — it could be **0**, **1**, or **infinite values**, **synchronously or asynchronously**.

### 🧠 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.

### 🧰 Where/When?

* API calls (HttpClient.get() returns an Observable)
* Form inputs (value changes)
* Real-time features like chat, stock prices
* Custom data streams like WebSockets

### 🔤 Example:

import { Observable } from 'rxjs';

const observable = new Observable<string>((observer) => {

observer.next('Hello');

observer.next('World');

observer.complete();

});

#### 🧾 Explanation:

* Observable<string>: The observable will emit string values.
* (observer) => { ... }: This is the function that will run when someone subscribes.
* observer.next(...): Sends a value to subscribers.
* observer.complete(): Marks the stream as complete. No more values can be emitted after this.

observable.subscribe({

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

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

});

#### 🧾 Explanation:

* subscribe(...): Starts listening to the observable.
* next: Called for each emitted value (Hello, then World).
* complete: Called when the observable is done emitting values.

### 🟢 Output:

Hello

World

Done!

## 👀 2. Observer

An **Observer** is just a plain object with up to three callback functions:

* next(value) – receives values.
* error(err) – handles errors.
* complete() – handles completion.

### When to define it explicitly?

* When you want better **separation of logic** (e.g., define the observer in one place, use it in another).
* In complex data pipelines or libraries.

## 🔗 3. Subscription

A **Subscription** is what you get when you subscribe to an Observable.

const subscription = observable.subscribe(observer);

subscription.unsubscribe(); // Cancels it

### 🔍 Why?

Useful when you:

* Need to **cancel API calls** or **unsubscribe from value changes**.
* Prevent **memory leaks** in Angular components (ngOnDestroy()).

## 🔄 4. Subject – A Multicasting Observable

### 🔍 What is it?

A **Subject** is both an Observable and an Observer. You can emit values and subscribe to them.

import { Subject } from 'rxjs';

const subject = new Subject<number>();

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

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

subject.next(1);

### 🧾 Explanation:

* Two subscribers: A and B.
* When next(1) is called, both subscribers receive the same value.
* 🔁 Use case: **Shared state** across components or services.

## 🔁 5. BehaviorSubject – With Initial Value

Unlike Subject, **BehaviorSubject** needs an **initial value** and **remembers the last emitted value**.

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

bSubject.next(42);

bSubject.subscribe(val => console.log(val)); // Output: 42

### 🔍 When to use it?

* Global state in Angular (AuthService, ThemeService)
* Form values with reactive forms
* When new subscribers need the **current value**

## 🧪 Real-world Scenario: HTTP + Observables in Angular

### Scenario: Fetch user list on component load.

// user.service.ts

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

import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })

export class UserService {

constructor(private http: HttpClient) {}

getUsers(): Observable<User[]> {

return this.http.get<User[]>('/api/users');

}

}

// user.component.ts

export class UserComponent implements OnInit {

users: User[] = [];

constructor(private userService: UserService) {}

ngOnInit() {

this.userService.getUsers().subscribe({

next: data => this.users = data,

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

});

}

}

### Explanation:

* getUsers() returns an Observable from HttpClient.get.
* We subscribe in ngOnInit() to get the data.
* Good practice: Use unsubscribe() in ngOnDestroy() or use async pipe in templates.

Perfect! Let’s continue by exploring the **most commonly used RxJS operators** — starting with **Transforming**, **Filtering**, **Combining**, **Error Handling**, and **Utility** operators — each explained in depth with **real-world Angular use cases**, **why/when to use them**, and **line-by-line code analysis**.

**🛠️ RxJS Operators – Detailed Guide with Examples**

**🔄 1. map() – Transform Values**

**📌 What It Does:**

Transforms each value emitted by an Observable using a function.

**✅ When to Use:**

* Format or preprocess data before using it
* Modify server responses
* Extract only the necessary fields

**💡 Real-World Angular Example:**

You receive user data from an API and want to extract only names.

import { of } from 'rxjs';

import { map } from 'rxjs/operators';

of({ name: 'Alice' }, { name: 'Bob' }).pipe(

map(user => user.name)

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

**🧾 Line-by-Line Explanation:**

of({ name: 'Alice' }, { name: 'Bob' }) // Emits two objects

.pipe(

map(user => user.name) // Transforms each object to just the name

)

.subscribe(name => console.log(name)); // Output: "Alice", then "Bob"

**🚿 2. filter() – Filter Values**

**📌 What It Does:**

Emits only those values that pass the given condition.

**✅ When to Use:**

* Filter user input (e.g., form values)
* Ignore unwanted data in a stream

**💡 Real-World Angular Example:**

Display users older than 18.

import { from } from 'rxjs';

import { filter } from 'rxjs/operators';

const users = [

{ name: 'Tom', age: 16 },

{ name: 'Jerry', age: 25 },

];

from(users).pipe(

filter(user => user.age >= 18)

).subscribe(user => console.log(user.name));

**🧾 Explanation:**

* Filters out users with age < 18
* Only "Jerry" is printed

**⏳ 3. debounceTime(ms) – Wait Before Emitting**

**📌 What It Does:**

Waits for a specified time after the last emission before actually emitting the value.

**✅ When to Use:**

* Search input boxes (debouncing typing)
* Prevent rapid calls to API

**💡 Real-World Angular Example:**

API search when the user stops typing.

import { fromEvent } from 'rxjs';

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

fromEvent(inputElement, 'input').pipe(

debounceTime(300), // Wait 300ms after last keystroke

map((event: any) => event.target.value)

).subscribe(value => {

// Call your API here

});

**🧾 Explanation:**

* Prevents firing an API call on every keystroke
* Common in search bars

**🔁 4. switchMap() – Switch to New Observable**

**📌 What It Does:**

Cancels previous inner Observable when a new one is emitted.

**✅ When to Use:**

* Typeahead search where only the latest request matters
* Cancelling previous API calls

**💡 Real-World Angular Example:**

User types in a search box → call search API.

searchInput$.pipe(

debounceTime(300),

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

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

**🧾 Explanation:**

* switchMap cancels the previous request if a new term is typed
* Only the latest request is handled
* Improves performance and UX

**🔀 5. mergeMap() – Run All Observables in Parallel**

**📌 What It Does:**

Maps to inner Observables and subscribes to all concurrently.

**✅ When to Use:**

* When you want **all** requests/responses to run in parallel
* E.g., loading multiple resources at once

**💡 Real-World Angular Example:**

Fetch multiple user profiles at once.

from([1, 2, 3]).pipe(

mergeMap(id => this.http.get(`/api/users/${id}`))

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

**🧾 Explanation:**

* Sends 3 parallel HTTP requests
* All responses are handled as they arrive

**📛 6. catchError() – Handle Errors Gracefully**

**📌 What It Does:**

Intercepts an error and returns a fallback Observable.

**✅ When to Use:**

* Prevent app crashes from failed HTTP requests
* Show error messages

**💡 Real-World Angular Example:**

Handle failed user load.

this.http.get('/api/user').pipe(

catchError(error => {

console.error('Error loading user', error);

return of({ name: 'Default User' }); // fallback value

})

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

**🧾 Explanation:**

* Catches HTTP error
* Returns a default object to keep the app running

**🧼 7. tap() – Side Effects (Logging)**

**📌 What It Does:**

Performs side effects like logging, without changing the stream.

**✅ When to Use:**

* Debug data flows
* Trigger analytics events or logs

**💡 Real-World Angular Example:**

this.http.get('/api/user').pipe(

tap(data => console.log('Fetched user:', data))

).subscribe();

**🧾 Explanation:**

* Logs data before it reaches the final subscriber
* Doesn’t alter the data itself

**🔚 8. finalize() – Cleanup Logic**

**📌 What It Does:**

Executes code when observable completes or errors out.

**✅ When to Use:**

* Hide loading spinners
* Close connections

**💡 Real-World Angular Example:**

this.http.get('/api/user').pipe(

finalize(() => console.log('Request completed'))

).subscribe();

**🗂 Summary Table of Common Operators**

| **Operator** | **Purpose** | **Common Use Case** |
| --- | --- | --- |
| map | Transform emitted values | Format API response |
| filter | Remove unwanted values | Ignore invalid user input |
| debounceTime | Delay emissions | API search optimization |
| switchMap | Cancel previous request | Typeahead / auto-suggest |
| mergeMap | Run inner Observables | Parallel HTTP requests |
| catchError | Handle and recover from errors | Show fallback UI or default values |
| tap | Log or trigger effects | Debug, analytics |
| finalize | Cleanup after complete/error | Hide spinners |

Great! Since you're using **Angular 17+ (Standalone APIs)** — likely Angular 18 or 19 — you no longer need **NgModules** like AppModule. Instead, you use **standalone components** and directly import HttpClient via provideHttpClient().

Let’s rewrite the Angular HTTP setup **the modern way (Angular 17–19)** using standalone APIs:

**🚀 Angular 19 HTTPClient with Standalone APIs**

**✅ 1. Enable HTTP in your App (No NgModules!)**

In Angular 15+, you use provideHttpClient() from @angular/common/http.

**✅ main.ts**

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

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

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

bootstrapApplication(AppComponent, {

providers: [provideHttpClient()]

});

**🧠 2. Create a Service with HttpClient**

No change here — services still work the same, but can be **provided at the root** without modules.

**✅ user.service.ts**

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

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

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

export interface User {

id: number;

name: string;

}

@Injectable({ providedIn: 'root' })

export class UserService {

constructor(private http: HttpClient) {}

getUsers(): Observable<User[]> {

return this.http.get<User[]>('/api/users').pipe(

catchError(err => {

console.error('Error fetching users:', err);

return of([]); // fallback empty list

})

);

}

}

**🧩 3. Use HTTP in a Standalone Component**

**✅ app.component.ts**

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

import { CommonModule } from '@angular/common';

import { UserService, User } from './user.service';

@Component({

selector: 'app-root',

standalone: true,

imports: [CommonModule],

template: `

<h1>Users</h1>

<p \*ngIf="error">{{ error }}</p>

<ul>

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

</ul>

`

})

export class AppComponent implements OnInit {

users: User[] = [];

error = '';

constructor(private userService: UserService) {}

ngOnInit() {

this.userService.getUsers().subscribe({

next: (data) => this.users = data,

error: () => this.error = 'Failed to load users'

});

}

}

**📦 Summary for Angular 19 (Standalone)**

| **Feature** | **Old (NgModules)** | **New (Standalone)** |
| --- | --- | --- |
| Provide HttpClient | HttpClientModule | provideHttpClient() in main.ts |
| Component setup | imports: [HttpClientModule] | standalone: true + imports: [] |
| Bootstrapping | NgModule.bootstrap | bootstrapApplication() |

**✅ Optional: Async Pipe (No Manual Subscription)**

Use the async pipe to auto-subscribe and handle updates in the template:

**Update user.service.ts**

getUsers(): Observable<User[]> {

return this.http.get<User[]>('/api/users');

}

**Update app.component.ts**

template: `

<h1>Users (Async)</h1>

<ul>

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

</ul>

`

export class AppComponent {

users$ = this.userService.getUsers();

constructor(private userService: UserService) {}

}

Excellent! Let’s continue step by step with real-world examples using **Angular 19 + Standalone APIs + RxJS + HttpClient**.

**📌 Part 1: switchMap with HttpClient (Typeahead Search)**

**✅ Use Case:**

You're building a **search box**. Each time the user types, you want to call the API, but cancel any ongoing request if the user keeps typing — to avoid sending outdated queries.

**🧠 Why switchMap?**

* Cancels the previous API request.
* Only keeps the latest request.
* Perfect for **auto-complete**, **live search**, or **form input** scenarios.

**🧪 Step-by-step Implementation**

**✅ 1. Component Setup**

// app.component.ts

import { Component, signal, effect, inject } from '@angular/core';

import { CommonModule } from '@angular/common';

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

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

import { fromEvent, map } from 'rxjs';

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

@Component({

selector: 'app-root',

standalone: true,

imports: [CommonModule, FormsModule],

template: `

<h1>🔍 Search</h1>

<input type="text" placeholder="Search..." #searchBox>

<ul>

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

</ul>

`

})

export class AppComponent {

results: string[] = [];

private http = inject(HttpClient);

ngAfterViewInit() {

const searchBox = document.querySelector('input')!;

fromEvent(searchBox, 'input').pipe(

map((event: any) => event.target.value),

debounceTime(300),

distinctUntilChanged(),

switchMap(query => this.http.get<string[]>(`/api/search?q=${query}`))

).subscribe({

next: data => this.results = data,

error: () => this.results = ['Error fetching data']

});

}

}

**🔍 Line-by-Line Breakdown**

fromEvent(searchBox, 'input')

* Creates a stream of input events (whenever the user types).

map((event: any) => event.target.value)

* Extracts the search text from the event.

debounceTime(300)

* Waits 300ms after the last keypress before continuing.

distinctUntilChanged()

* Ignores duplicate queries.

switchMap(query => this.http.get<string[]>(`/api/search?q=${query}`))

* Cancels the previous API call if a new character is typed.
* Makes a GET request to your server.

Great choice! 🎯

In **Angular**, **interceptors** are powerful tools to **add headers (like auth tokens)**, **log HTTP requests**, **handle global errors**, and more — all in **one centralized place**.

**🔐 Part 2: Set Custom Headers & Auth Tokens with Interceptors (Angular 19+)**

**📌 Use Case:**

You have a token stored (e.g., JWT or access token), and you want to:

* Automatically attach it to **every** HTTP request
* Avoid manually setting headers in each service
* Handle token expiration or unauthorized access globally

**🧠 Why Use Interceptors?**

* Reusable and centralized
* Avoid header duplication in every request
* Ideal for authorization, logging, retries, error handling

**✅ Step-by-Step Setup (Standalone API Compatible)**

**✅ 1. Create the Interceptor**

// auth.interceptor.ts

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

export const authInterceptor: HttpInterceptorFn = (req, next) => {

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

if (token) {

const authReq = req.clone({

setHeaders: {

Authorization: `Bearer ${token}`

}

});

return next(authReq);

}

return next(req);

};

**🧾 Explanation:**

* HttpInterceptorFn: Angular 16+ function-style interceptor
* req.clone(...): Never mutate the original request; clone it
* Adds Authorization: Bearer <token> header
* next(authReq): Forwards the modified request

**✅ 2. Register Interceptor in main.ts**

// main.ts

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

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

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

import { authInterceptor } from './app/auth.interceptor';

bootstrapApplication(AppComponent, {

providers: [

provideHttpClient(

withInterceptors([authInterceptor])

)

]

});

**✅ 3. Use HttpClient Anywhere — Header Is Already Added**

// user.service.ts

getProfile(): Observable<User> {

return this.http.get<User>('/api/profile'); // No need to set headers!

}

✅ Automatically attaches Authorization header  
✅ Works with **all** HttpClient requests  
✅ Great for securing API calls

**🧪 Bonus: Add Content-Type or Custom Headers**

const authReq = req.clone({

setHeaders: {

Authorization: `Bearer ${token}`,

'X-App-Version': '1.0.0',

'Content-Type': 'application/json'

}

});

**❗ Optional: Handle 401 Unauthorized Globally**

Extend the interceptor to handle token expiration:

return next(authReq).pipe(

catchError(err => {

if (err.status === 401) {

console.warn('Unauthorized! Redirecting to login...');

// Optionally: router.navigate(['/login']);

}

return throwError(() => err);

})

);

**✅ Summary: Interceptors with Standalone Angular**

| **Purpose** | **Code Pattern** |
| --- | --- |
| Add Auth Token | req.clone({ setHeaders }) |
| Register Interceptor | withInterceptors([...]) in main.ts |
| Handle Errors Globally | Use catchError() in the interceptor |