Sistemi Web Lezione 2

Direttive · Pipe · Services/DI · RxJS (focus)

Direttive · Pipe · Services/DI · RxJS (focus)

Obiettivi:

- Capire e usare direttive strutturali/attributo
- Formattare l'UI con Pipe (incl. `async`)
- Progettare servizi con Depedency injection (`inject`)
- **Pensare in RxJS**: operatori, errori, combinazioni

Modulo 1 — Direttive

Cosa sono, quando usarle, nuovi costrutti di controllo

Direttive

- Una **direttiva** è una classe che aggiunge **comportamento** o **struttura** al DOM.
- **Strutturali**: modificano il layout aggiungendo/rimuovendo nodi (es. `*ngIf`, `*ngFor`, `*ngSwitch` — moderni: `@if`, `@for`).
- **Attributo**: cambiano aspetto/comportamento di un elemento (es. `[ngClass]`, `[ngStyle]`, `ngModel`, direttive custom).
- **Uso tipico**: show/hide, iterazioni, gestione classi/stili, a11y con `aria-*`, riuso di logica nel template.
- **Buone pratiche**: template leggeri, funzioni pure, usare `trackBy` / `track` nelle liste, direttive piccole e focalizzate.

Direttive — esempio

```
export interface Product { id: string; title: string; price: number; }
products: Product[] = [
  { id: 'p1', title: 'Runner Pro', price: 89.9 },
  { id: 'p2', title: 'Cappellino',  price: 19.9 }
];
showSaleOnly = false;
toggleSaleOnly(){ this.showSaleOnly = !this.showSaleOnly; }
get visible(): Product[] { return this.showSaleOnly ? this.products.filter(p=>p.price<20) : this.products; }
```

```
<button (click)="toggleSaleOnly()">
  {{ showSaleOnly ? 'Mostra tutti' : 'Solo sconti' }}
</button>
<ul>
  <li *ngFor="let p of visible" [class.sale]="p.price < 20">
    {{ p.title }} — {{ p.price | currency:'EUR' }}
  </li>
</ul>
<p *ngIf="!visible.length">Nessun prodotto</p>
```

Esempio — `@if`

```
<section>
  @if (product?.onSale) {
    <span class="badge">Sconto</span>
  } @else {
    <span aria-hidden="true"></span>
  }
</section>
```

`@if` vs `*ngIf`

Quando preferire `@if`:

- Blocchi espliciti `{ }` → meno ambiguità
- `@else if` nativo → rami multipli puliti
- Miglior supporto per template complessi
- Binding `; as x` supportato nel blocco
- Strutture multiple più leggibili

Prima/DOPO — `*ngIf` vs `@if`

```
<!-- Prima -->
<div *ngIf="vm$ | async as vm; else loading">
  <h3>{{ vm.title }}</h3>
</div>
<ng-template #loading>Caricamento…</ng-template>
```

```
<!-- Dopo -->
@if (vm$ | async; as vm) { <h3>{{ vm.title }}</h3> }
@else { Caricamento… }
```

`else if` con `@if`

```
@if (status==='ok') { OK }
@else if (status==='warn') { Attenzione }
@else { Errore }
```

Costrutto for

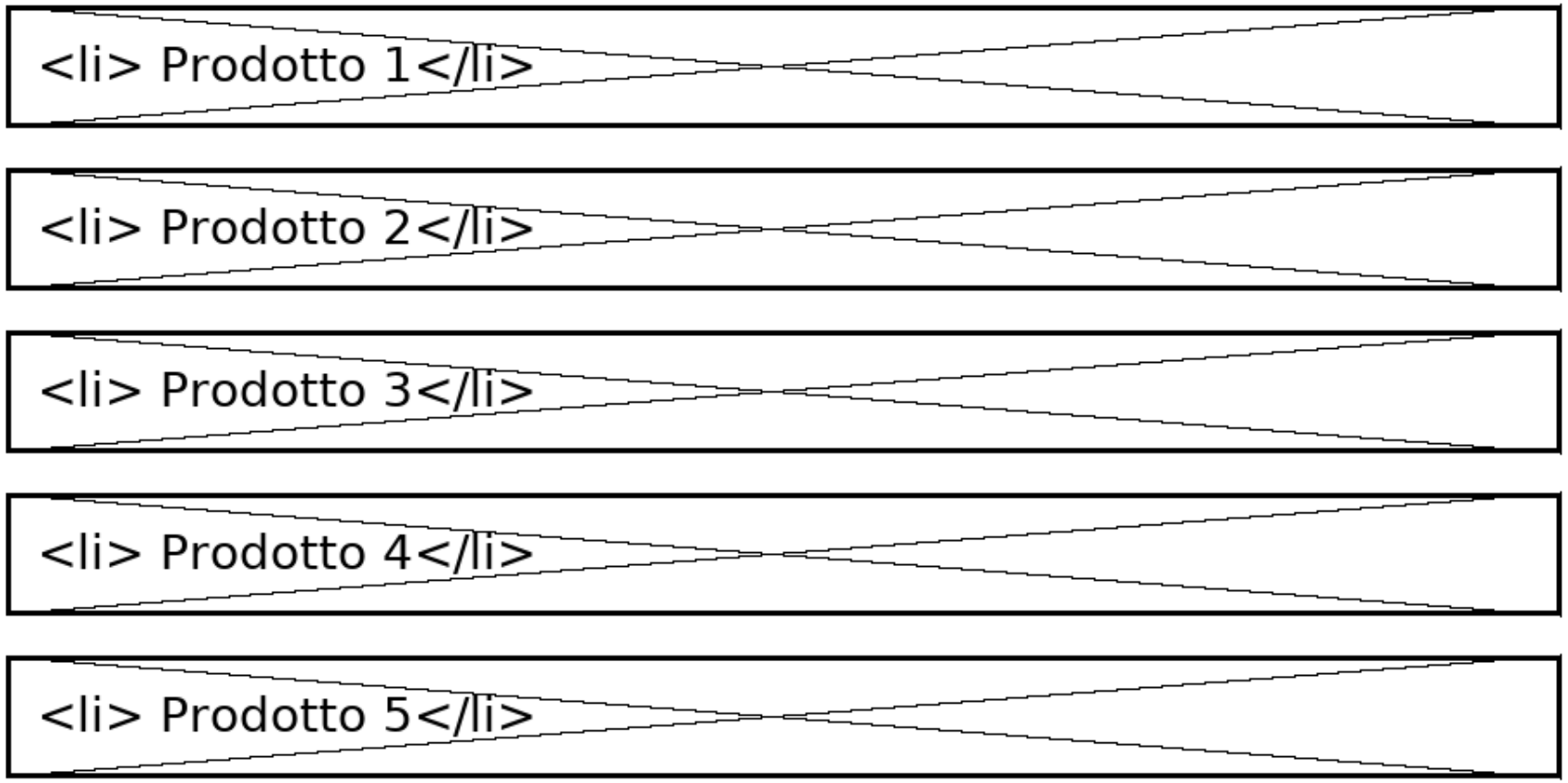
```
<!-- Dopo -->
@for (p of products; track p.id; let i = $index) {
  <li>{{ i+1 }}. {{ p.name }}</li>
} @empty { Nessun prodotto }
```

Costrutto for - Approfondimento track / trackBy

- Con liste grandi o riordinamenti, **il default può ricreare nodi DOM**.
- track/trackBy aiuta Angular a riusare gli elementi.
- Viene tracciato un identificatore stabile (es. id) per ogni elemento → evita re-render e perdita di stato (focus, input).
- Usarli su liste medie/grandi, riordinamenti/filtri frequenti, item con componenti costosi.
- **Non usare mai** l'indice come chiave (si rompe su riordino).

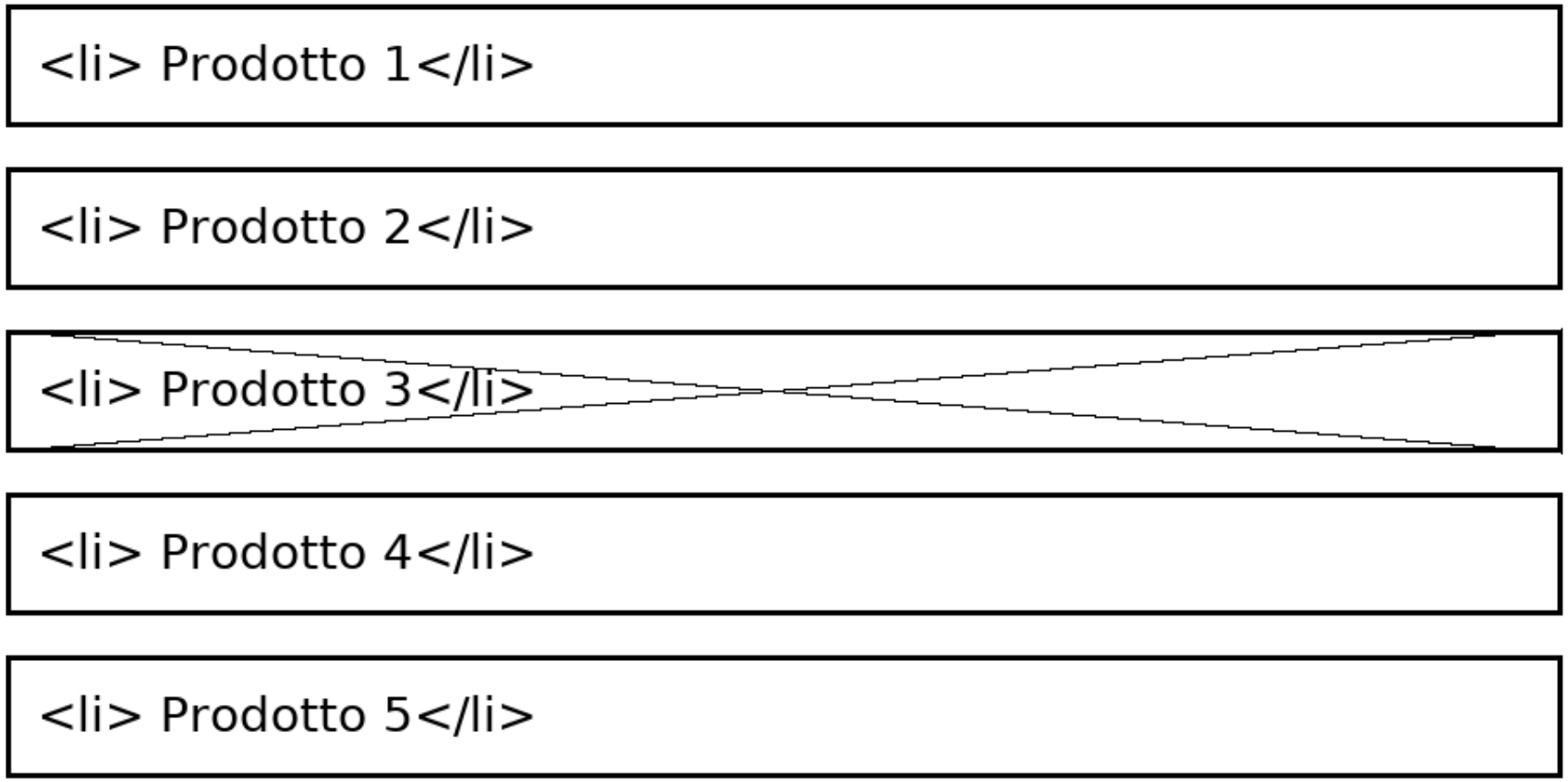
`track` prima/dopo

Prima: lista senza `track`



`track` prima/dopo

Dopo: lista con `track p.id`



`*ngFor` vs `@for` — differenze chiave

- `trackBy` funzione → **`track`** **espressione** (p.id)
- Variabili locali: `let i = index`, ecc.
- Blocco **`@empty`** per liste vuote
- DOM più stabile (focus/scroll preservati)

Prima/DOPO — `*ngFor` → `@for`

```
<!-- Prima -->
<li *ngFor="let p of products; trackBy: trackById">
  {{ p.name }}
</li>
```

```
<!-- Dopo -->
@for (p of products; track p.id; let i = $index) {
  <li>{{ i+1 }}. {{ p.name }}</li>
} @empty { Nessun prodotto }
```

`*ngSwitch` vs `@switch` — differenze chiave

- Blocchi per `@case` e `@default`
- Sintassi coerente con `@if/@for`
- No “fallthrough”: ogni case è isolato

Prima/DOPO — `*ngSwitch` → `@switch`

```
<!-- Prima -->
<div [ngSwitch]="role">
  <p *ngSwitchCase="'admin'">Admin</p>
  <p *ngSwitchDefault>Guest</p>
</div>
```

```
<!-- Dopo -->
@switch (role) {
  @case ('admin') { <p>Admin</p> }
  @default { <p>Guest</p> }
}
```

Cheatsheet

- `*ngIf="cond; else tpl"` → `@if (cond) { … } @else { … }`
- `*ngIf="obs | async as x"` → `@if (obs | async; as x) { … }`
- `*ngFor="let x of xs; trackBy: t"` → `@for (x of xs; track x.id)`
- Lista vuota → `@empty { … }`
- `[ngSwitch]` / `*ngSwitchCase` → `@switch` / `@case` / `@default`

Edge cases & best practice

- Evita `track` con indice → **usa id stabile**
- Usa `@empty` per messaggi a11y (contenitore con `aria-live="polite"`)
- Acquisisci valori con `; as value` (no subscribe nel TS)
- Mantieni i blocchi brevi → logica nel TS/servizi

Direttive di attributo utili

- `[ngClass]`, `[ngStyle]` per stato/tema
- `ngModel` solo in casi semplici (teaser L4: Reactive Forms)
- A11y: classi per focus visibile, contrasto sufficiente

Direttive di comportamento

Le direttive di comportamento si ancorano a un elemento già esistente e ne **modificano l'aspetto o la reattività** senza cambiare la struttura del DOM. Sono perfette quando vuoi **aggiungere un comportamento trasversale** (es. focus, visibilità, stile reattivo, gestione eventi) senza introdurre un nuovo componente o duplicare markup.
Possono essere definiti come "micro-plugin" riusabili, con un'API piccola e chiara.

Built-in utili

`[ngClass]` e `[ngStyle]` sono versatili: vanno usate per **stato visivo** reattivo, mantenendo la logica nel TS.

`ngModel` va bene per **casi semplici** e demo; per validazioni serie preferisci **Reactive Forms**.

Il trucco è **non far crescere i binding**: se vedi tante condizioni annidate, sposta la decisione in una proprietà (Es. CSS).

```
<!-- 1–2 binding, non di più -->
<div [ngClass]="{ 'on-sale': p.onSale, 'low-stock': p.stock<5 }">
  {{ p.name }}
</div>
```

Progettare una direttiva custom (linee guida)

Una buona direttiva espone **pochi input ben tipizzati**, evita side-effect imprevisti e comunica l'intento nel nome (`autofocus`, `scrollShadow`). Non nasconde logica di dominio; si limita a **orchestrare stile/attributi/eventi**.

Una direttiva può implementare hook (es. `AfterViewInit`) e usare `@HostBinding` / `@HostListener`.

Esempio 1 — `AutofocusDirective`

Una direttiva sobria che mette il focus sul campo al montaggio del componente, senza toccare il template.

```
@Directive({ selector: '[appAutofocus]', standalone: true })
export class AutofocusDirective implements AfterViewInit {
  constructor(private el: ElementRef<HTMLInputElement>) {}
  ngAfterViewInit(){ queueMicrotask(()=> this.el.nativeElement.focus()); }
}
```

```
<input type="text" appAutofocus>
```

Esempio 2 — `ScrollShadowDirective` (TS ≤12)

Attiva/disattiva una classe in base allo scroll dell'elemento host (utile su toolbar o liste).

```
@Directive({ selector:'[appScrollShadow]', standalone:true })
export class ScrollShadowDirective {
  @HostBinding('class.has-shadow') shadow = false;
  @HostListener('scroll', ['$event.target']) onScroll(el: HTMLElement){
    this.shadow = el.scrollTop > 0;
  }
}
```

```
<div appScrollShadow class="toolbar scrollable">…</div>
```

Input “robusti”: trasformazioni

Inserisci riferimenti chiari per il chiamante: **trasforma l’input** nel tipo corretto.

Per booleani/numero usa i transformer built-in: il codice resta più sicuro e leggibile.

```
@Input({ transform: booleanAttribute }) appActive = false;
@Input({ transform: numberAttribute  }) appDelay  = 0;
```

In questo modo `appActive="true"` o `appActive=""` diventano realmente `true`, e `appDelay="250"` diventa un numero.

A11y e direttive: cosa non rompere

Una direttiva non deve **spezzare la semantica**: non rimuovere attributi nativi come `role`, `aria-*`, `tabindex` se non ne si e' veramente sicuri.
Se si cambia la visibilità, **aggiorna anche ARIA** (`aria-hidden`, `aria-live`).
Se gestisci focus, **rispetta l'ordine tab** e fornisci sempre un fallback via tastiera.

Errori comuni (e alternative)

Quando vedi **molta logica nelle condizioni CSS** o listener complessi, stai costruendo un componente mascherato: estrai un vero componente.

Evita direttive che cambiano layout e dati.

Per side-effect di rete o business, usa un **service**; la direttiva si limita a reagire all'UI.

Recap Modulo 1

Takeaway:

1. Strutturali vs attributo: responsabilità diverse
2. `@if/@for` migliorano leggibilità
3. Usa `track` sempre con liste

Domande:

- Quando preferire `@if`?
- Perché non usare l'indice come chiave?

Cosa sono le pipe

Le Pipe vivono **accanto alla UI**: prendono un valore e lo **rendono parlante** per l'utente finale, senza toccare la logica di dominio.

Possono essere pensate come l'ultimo miglio tra il tuo modello e l'occhio umano: formattano numeri, date, stringhe.

Quando la trasformazione è solo visiva, la Pipe è il luogo giusto; quando nasconde decisioni di business, è il posto sbagliato.

Pipe “pure”

Una Pipe “pure” ricalcola **solo** se cambiano i suoi input; questo la rende prevedibile e veloce. In un’app reattiva dove i dati arrivano a stream, il cambio input è l’evento principale.

Le Pipe “impure” invece si rieseguono a ogni ciclo di change detection: sono utili in casi particolari, ma sono molto costose.

Se ti serve reattività, **crea lo stream nel TS** e usa `async`, invece di forzare una Pipe impura.

Locale e formattazione

`currency`, `number`, `percent` e `date` non sono solo estetica: rispettano **regole locali** (separatore decimale, simboli, formato data).

Ad esempio in italiano, `EUR` e un formato data “mediumDate” rendono immediata la lettura. La UX migliora perché l’utente non deve “tradurre mentalmente” i dati.

```
<p>Prezzo: {{ p.price | currency:'EUR' }}</p>
<p>Creato il: {{ p.createdAt | date:'mediumDate' }}</p>
```

`AsyncPipe`: il ponte sicuro tra stream e template

`async` si occupa di **sottoscrivere e rilasciare** automaticamente `Observable` e `Promise`.

Il template resta dichiarativo, i componenti restano snelli e non scriviamo `ngOnDestroy`.

Quando i pezzi di UI dipendono da più stream, comporli nel TS e proiettarli con un **`vm$` unico** evita accoppiamenti fragili nel template.

```
<!-- Proietta un singolo valore “derivato” -->
<span>Articoli economici: {{ cheapCount$ | async }}</span>
```

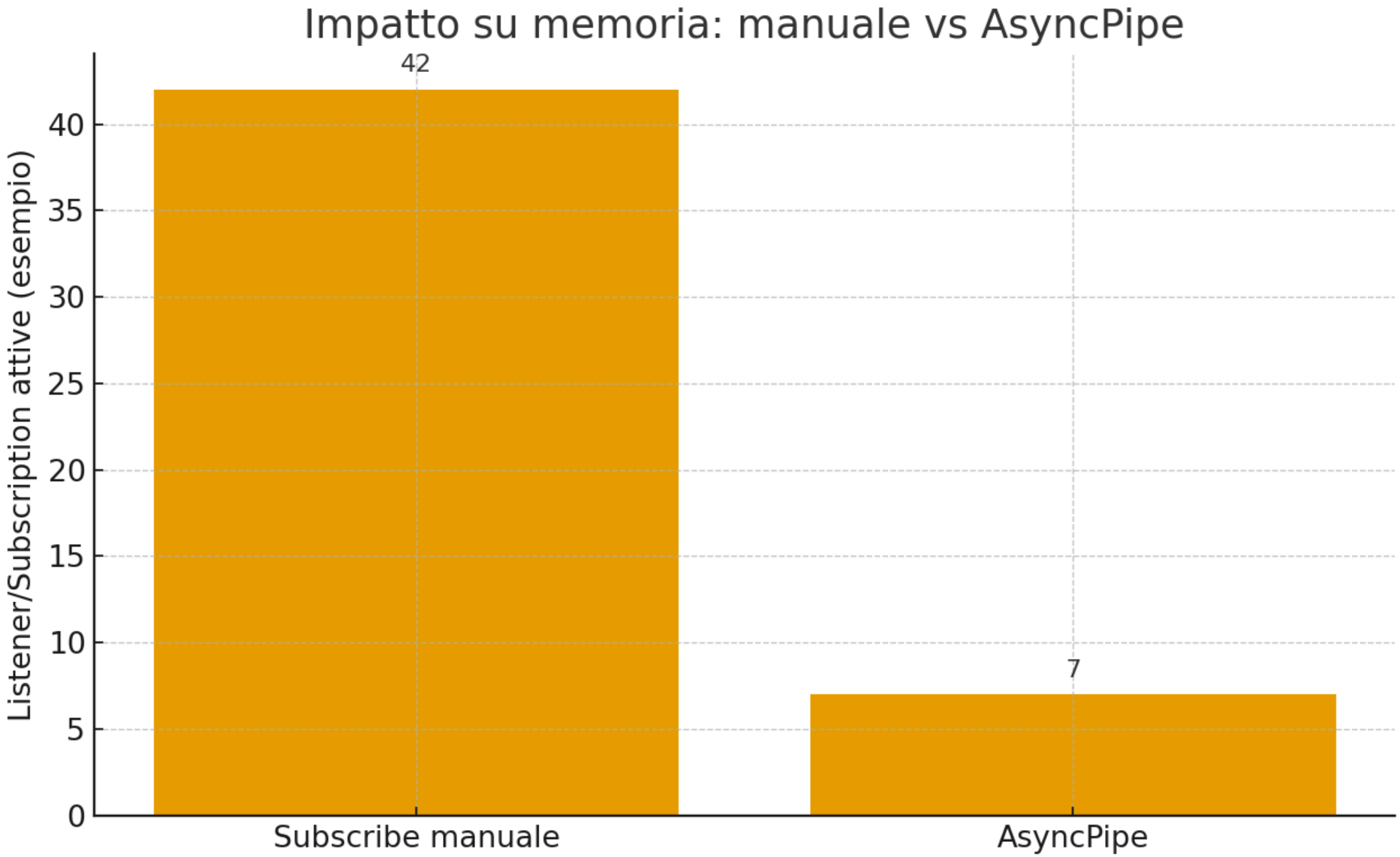
Prima/Dopo — subscribe manuale → `async`

Prima (anti-pattern)

subscribe in `ngOnInit` + assegnazione + unsubscribe manuale

Dopo

`products$` + `async` nel template (nessun cleanup manuale)


Impatto su memoria: manuale vs AsyncPipe
Listener/Subscription attive (esempio)
0
5
10
15
20
25
30
35
40
42
7
Subscribe manuale
AsyncPipe

Pipe built-in principali

- `currency` per i prezzi
- `date` per la cronologia
- `percent` per sconti
- `titlecase` per titoli coerenti
- `slice` per anteprime brevi
- `keyvalue` è utile per ispezionare oggetti (debug o metadata), con comparatore opzionale quando l'ordine serve a qualcosa (menu, filtri)

```
<!-- keyvalue con comparatore -->
<div *ngFor="let e of info | keyvalue: (a,b) => a.key.localeCompare(b.key)">
  {{ e.key }}: {{ e.value }}
</div>
```

Evitare “logica in pipe”: dove tracciare il confine

Se la trasformazione **decide** cosa mostrare (business), non è più presentazione. In quel caso sposta la decisione nel servizio o nel componente e passa al template un valore già pronto. Le Pipe dovrebbero **formattare**, non orchestrare.

Questa separazione rende i test più facili e il codice più leggibile.

A11y con le Pipe: formattare per chi ascolta

Non tutto ciò che è gradevole a schermo è chiaro per uno screen reader. Valute e date possono richiedere un **`aria-label` descrittivo**. Se tagli un testo con una Pipe, assicurati che il contenuto completo sia raggiungibile (tooltip, `title`, dettagli).

```
<span [attr.aria-label]="'Prezzo: ' + (p.price | currency:'EUR')">
  {{ p.price | currency:'EUR' }}
</span>
```

Una Pipe custom: input tipizzato e null-safe

Le Pipe custom devono accettare **valori incerti** (null/undefined) ed essere **pure**.

Esempio di una `truncate` che gestisce quando il campo manca e rende chiaro il limite.

```
@Pipe({ name:'truncate', standalone:true })
export class TruncatePipe implements PipeTransform {
  transform(v: string | null | undefined, len=20){
    const s = v ?? '';
    return s.length > len ? s.slice(0, len) + '…' : s;
  }
}
```

```
<p>{{ p.description | truncate: 80 }}</p>
```

Prestazioni: dove mettere il calcolo

Se noti che un calcolo è **costoso** (ordinamenti, filtri, aggregazioni), spostalo nel TS dentro uno stream (`map` , `scan` , `combineLatest`) e proietta il risultato.

Le Pipe devono restare leggere; l'**AsyncPipe + RxJS** è il posto giusto per il lavoro pesante.

```
vm$ = combineLatest([products$, filters$]).pipe(
  map(([ps,f]) => ({ total: ps.length, cheap: ps.filter(p=>p.price<20).length }))
);
```

```
<!-- HTML -->
<p>Totali: {{ (vm$ | async)?.total }} — Economici: {{ (vm$ | async)?.cheap }}</p>
```

Pipe impure

Capita di dover riflettere cambiamenti che non cambiano l'**identità** dell'input (es. oggetti che mutano internamente).
Una Pipe impura può aggiornare l'output in ogni Change Detection, ma è una **scelta consapevole**: misura l'impatto e considera alternative (immutabilità, stream che emettono nuove istanze).

```
@Pipe({ name:'volatileMetrics', pure:false, standalone:true })
export class VolatileMetricsPipe implements PipeTransform {
  transform(stats: Metrics){ return stats.render(); }
}
```

Recap Pipe

- Le Pipe **formattano e chiariscono**; gli stream **decidono**. `async` unisce i due mondi senza perdere controllo di memoria.
- Locale e a11y non sono accessori, ma parte dell'esperienza.
- Le Pipe custom sono piccole, tipizzate e pure. Le impure si usano di rado, e solo dopo aver esaurito alternative più pulite.

Direttiva di comportamento vs componente/pipe

Se devi **rispettare il markup esistente** e solo “potenziarlo”, una direttiva è spesso la scelta giusta. Se invece vuoi **incapsulare UI e template**, un componente è più adatto; se devi **proiettare valori** (formattazione), una pipe è il posto giusto. La direttiva vince quando il bisogno è **cross-cutting**: stessa logica, tanti target diversi, zero duplicazioni.

Modulo 3 — Services & Dependency Injection

Separare logica/I·O dai componenti; DI moderna

Servizi

Un servizio è **il luogo della logica di applicazione** che non appartiene alla presentazione: orchestrazione di I/O, mapping dei dati, regole di business, logiche di cache.

Separando queste responsabilità dai componenti otteniamo **UI leggere**, codice **riusabile** e **testabile**.
Pensa ai componenti come "**proiettori**" (visualizzano) e ai servizi come "**motori**" (decidono, trasformano, chiamano).

Servizi

Un buon servizio espone una **API minima e coerente** (es. `list()`, `find(id)`, `search(q)`), ritorna **Observable tipizzati**, e non “svela” dettagli interni (es. non esporta `Subject` mutabili).

Gli oggetti in uscita sono **immutabili** o trattati come tali. Gli errori diventano **stati** mappati a monte, non `console.error`.

`inject()` vs costruttore:

`inject()` consente di ottenere dipendenze **fuori dal costruttore** (anche in funzioni).
È utile per servizi, guard, operatori e helper. Il costruttore resta ottimo nei componenti.

Usiamo quello che rende il codice **più leggibile**.

```
@Injectable({ providedIn:'root' })
export class ProductService {
  #http = inject(HttpClient);            // comodo in metodi statici/factory
  list(): Observable<Product[]> { return this.#http.get('/api/products'); }
}
```

Scope dei provider

- `providedIn:'root'`: un'unica istanza condivisa (config, API, cache brevi)
- **A livello componente (`providers:[...]`)**: istanza **isolata** per pagina/feature (stato locale)
- **A livello rotta/feature**: provider caricati solo quando servono (lazy load)

```
@Component({
  selector:'product-list',
  templateUrl:'./tpl.html',
  providers:[ProductListStore] // stato locale a questa pagina
})
export class ProductListComponent {}
```

Mini-store locale con servizio “scoped”

Un servizio “di pagina” incapsula **stato + azioni**; il componente consuma solo stream **readonly**.

```
@Injectable()
export class ProductListStore {

  #svc = inject(ProductService);
  #page = new BehaviorSubject(1);
  readonly page$ = this.#page.asObservable();
  readonly items$ = combineLatest([this.#svc.list(), this.page$]).pipe(
    map(([xs,p]) => xs.slice((p-1)*10, p*10))
  );
  setPage(p:number){ this.#page.next(p); }

}
```

InjectionToken

Per valori di configurazione usa `InjectionToken` **tipizzati** invece di costanti globali.

```
export const API_URL = new InjectionToken<string>('API_URL');
export const PAGE_SIZE = new InjectionToken<number>('PAGE_SIZE', { factory:()=>10 });

bootstrapApplication(App, {
  providers: [{ provide: API_URL, useValue:'/api' }]
});
```

```
class ProductService {
  #http = inject(HttpClient);
  #api = inject(API_URL);
  list(){ return this.#http.get<Product[]>(`${this.#api}/products`); }
}
```

Provider avanzati: `useClass`, `useValue`, `useExisting`, `useFactory`

Scegli la strategia di binding in base al caso d’uso (mock, alias, factory condizionale).

```
providers: [
  { provide: Logger, useClass: ConsoleLogger },              // implementazione concreta
  { provide: CURRENCY_CODE, useValue: 'EUR' },               // valore costante
  { provide: SearchService, useExisting: ProductService },   // alias
  { provide: ProductApi,
    useFactory: (env:Env)=> env.dev? new MockApi(): new HttpApi(),
    deps:[ENV_TOKEN] }
]
```

Service vs Component vs Pipe

- **Service**: decide, orchestra, chiama I/O, espone stream
- **Component**: riceve stream e **proietta** (nessun `subscribe` nel TS)
- **Pipe**: **formatta** per la vista (niente business logic)

Se ti trovi a manipolare DOM/stili in un servizio, probabilmente sei nel posto sbagliato → direttiva o componente.

Testabilità: servizi isolati

Servizi atomici o con dipendenze iniettate sono semplici da testare.

```
describe('ProductService', () => {
  beforeEach(()=> TestBed.configureTestingModule({
    providers:[ ProductService, {provide: API_URL, useValue:'/api-test'} ]
  }));
  it('list ritorna prodotti', (done)=>{
    const svc = TestBed.inject(ProductService);
    svc.list().subscribe(xs => { expect(xs.length).toBeGreaterThan(0); done(); });
  });
});
```

Anti-pattern DI (da evitare)

- **Unico servizio centralizzato**: conosce tutto, fa tutto → spezzalo per contesto
- **Svelare `Subject` mutabili**: esporre solo Observable
- **Creare servizi via `new`** (bypassa DI) → niente override/test
- **Leak di stato globale**: usa scope a componente per stati effimeri
- **Iniettare componenti nei servizi**: inverte i livelli

Checklist

- So descrivere in **una frase** la sua responsabilità?
- L’API è **minima** e tipizzata?
- Lo stato da mantenere è **davvero condiviso** (root) o locale (component)?
- Ho bisogno di un **token** per le configurazioni?
- Gli stream esposti sono **readonly** e mappati sugli errori?

Modulo 4 — RxJS

RxJS

Nelle nostre applicazioni, i dati **arrivano nel tempo**: richieste HTTP, input utente, timer, eventi UI. Vogliamo comporre queste sorgenti in modo **dichiarativo**, poter **cancellare** operazioni obsolete (es. ricerche), e tenere la UI **sempre allineata** con lo "stato di verità".

RxJS nasce per questo: trattare “valori nel tempo” come **primi cittadini**.

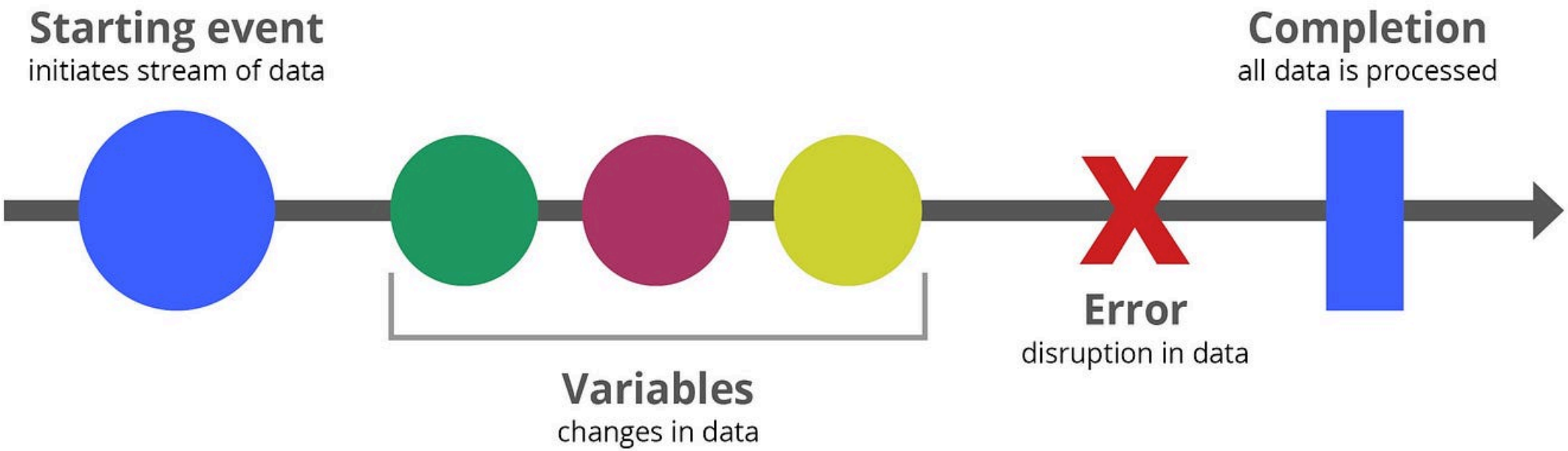
Programmazione reattiva

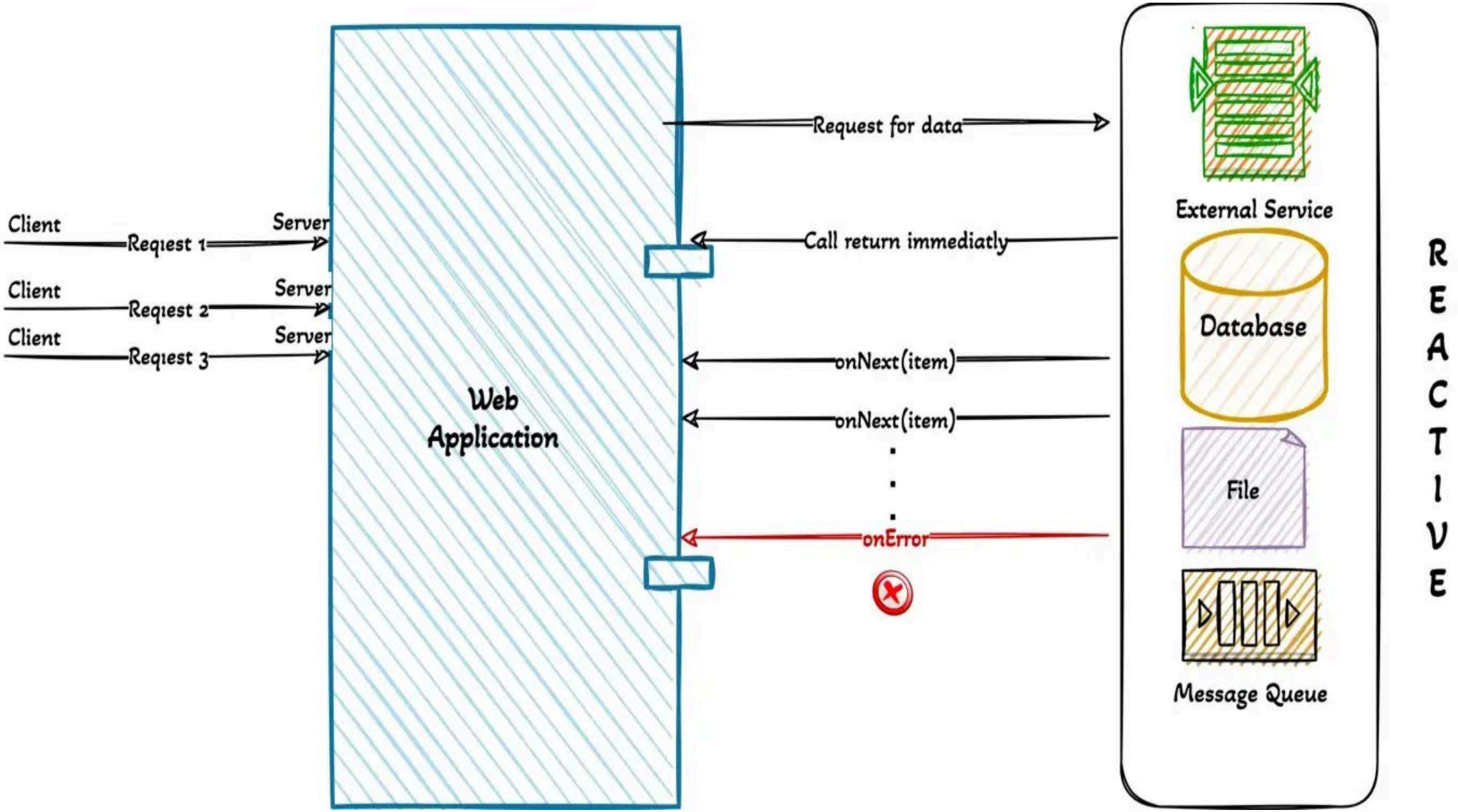
La programmazione reattiva sposta il paradigma da **pull** (chiedo io) a **push** (il dato mi “spinge” aggiornamenti).

Uno stream è una **sequenza ordinata nel tempo**: può emettere 0..n valori, per poi completare o fallire.

Componiamo stream con **operatori puri**; i side-effect vivono ai bordi, al **subscribe**.

Reactive programming uses asynchronous data streams




Client
Request 1
Server
Client
Request 2
Server
Client
Request 3
Server
Web Application
Request for data
Call return immediatly
onNext(item)
onNext(item)
onError
External Service
Database
File
Message Queue
REACTIVE

Dove RxJS semplifica

- **UI derivata**: filtro + sort + range prezzo + “solo sconti” → un unico `view$`
- **Cancellazione implicita**: `switchMap` su ricerche digitate
- **Stati di caricamento/errore**: `startWith` / `catchError` in un solo stream
- **Interop Angular**: `AsyncPipe` proietta il valore senza leak

Observable vs Promise

- **Promise**: un valore, non cancellabile, parte subito
- **Observable**:
  - 0..n valori,
  - **lazy**,
  - **cancellabile**,
  - si puo comporre con operatori

Se ti servono **più eventi** (input, scroll) o **annullare** richieste, l'Observable è la scelta naturale.
Angular **HttpClient** ritorna già Observable.

Observable

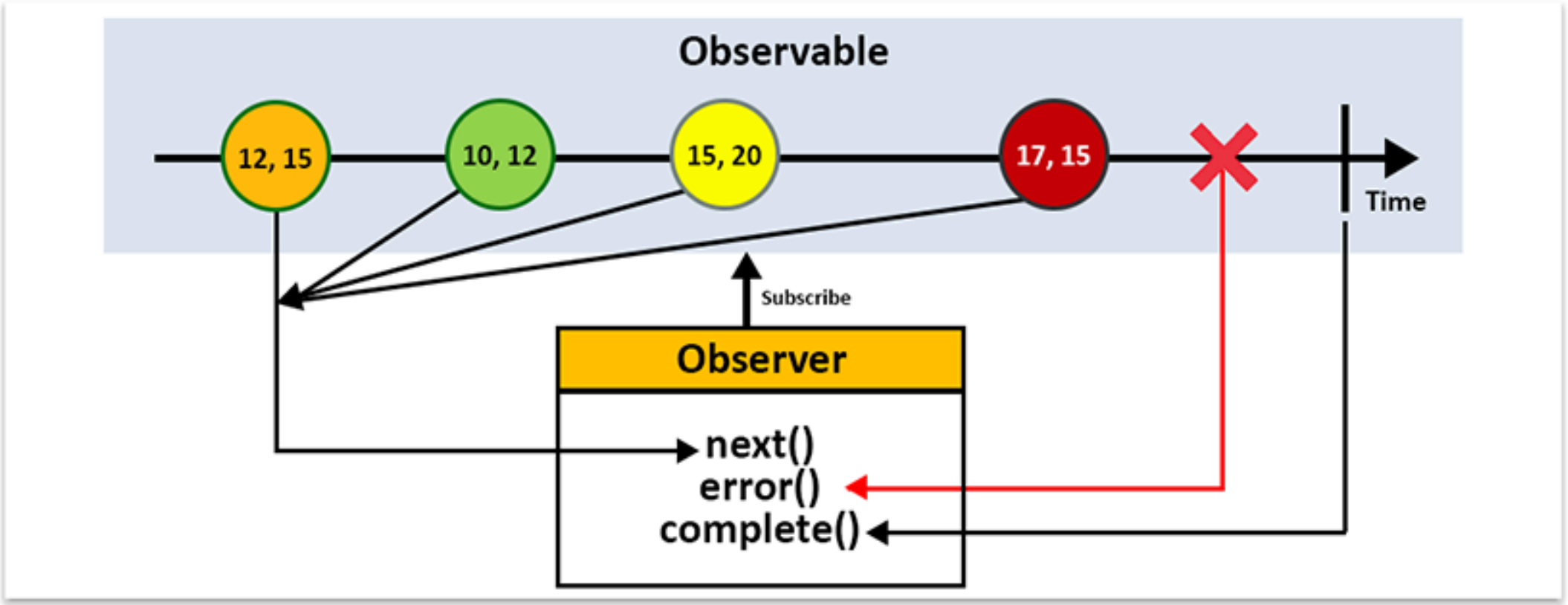
```
import { Observable } from 'rxjs';

const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});
```

Subscribe

```
observable.subscribe({
  next(x) {
    console.log('got value ' + x);
  },
  error(err) {
    console.error('something wrong occurred: ' + err);
  },
  complete() {
    console.log('done');
  },
});
console.log('just after subscribe');
```

```
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

Understanding RxJS and Observables in Angular



Creazione di stream

Altri modi per creare stream sono:

- `of`, `from`, `interval`, `fromEvent`
- Side-effect solo in **subscribe**
- Tipizza sempre gli stream

```
const a$ = of(1,2,3);
const click$ = fromEvent(buttonEl, 'click');
const prices$ = from(apiPrices()); // da Promise
```

Cold vs Hot

- **Cold**: ogni subscriber ha la “sua” sorgente (es. `http.get`, `of(...)`)
- **Hot**: la sorgente è condivisa e “scorre” (es. `fromEvent`, `Subject`)

Con i flussi hot ragioniamo su **multicast** e **condivisione** (cache, replay) in modo esplicito.

Subjects: segnali “hot”

- `Subject`: multicasting semplice
- `BehaviorSubject`: ultimo valore + default
- `ReplaySubject`: buffer storico

Pattern base: costruire un ViewModel$ unico

Componiamo tutte le sorgenti **fuori dal template** e pubblichiamo un solo stream tipizzato, pronto per la vista (loading/data/error). Il template resta minimale.

```
// TS ≤12 — VM unificato
vm$ = this.svc.list().pipe(
  map(data => ({ data, loading:false, error:null })),
  startWith({ data:[], loading:true, error:null }),
  catchError(err => of({ data:[], loading:false, error:err }))
);
```

```
@if (vm$ | async; as vm) {
  @if (vm.loading) { Caricamento… }
  @else if (vm.error) { Errore }
  @else { <!-- render della lista --> }
}
```

Operatori fondamentali

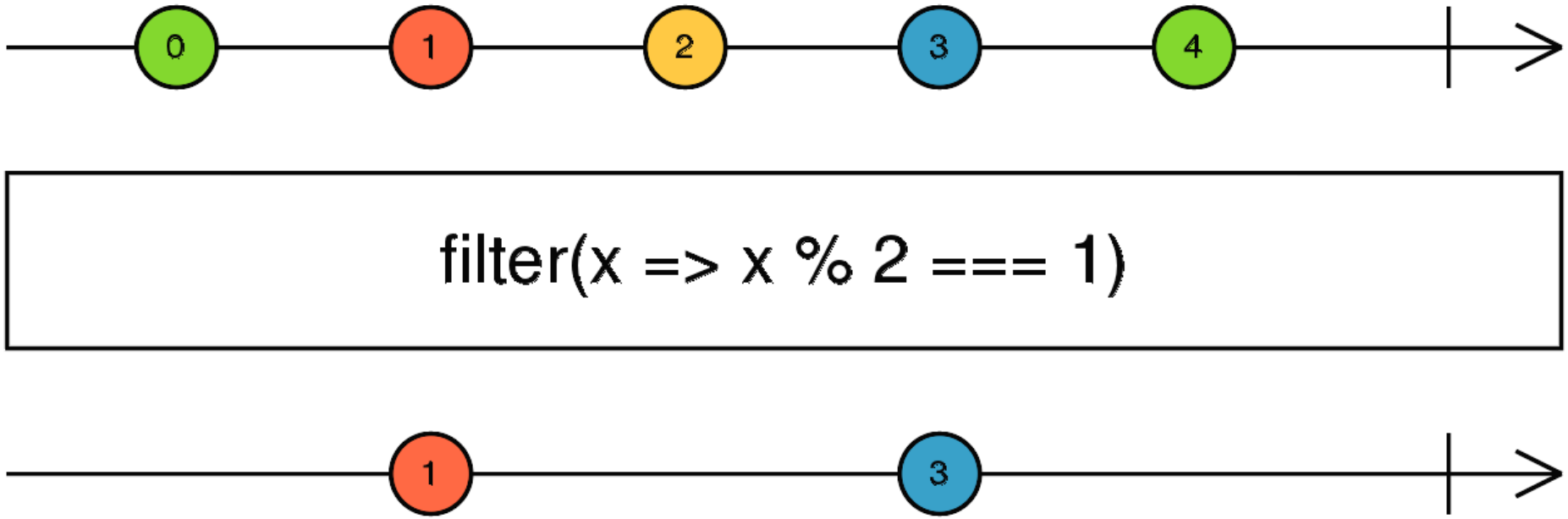
- **Trasformazione**: `map`, `filter`, `tap` (debug/side-effect)
- **Tempo**: `debounceTime`, `throttleTime`, `auditTime` (backpressure)
- **Combinazione**: `combineLatest`, `withLatestFrom`, `forkJoin`
- **Flattening**: `switchMap`, `concatMap`, `mergeMap`, `exhaustMap`

Trasformazione

- `map` Permette di mappare con un nuovo elemento
- `filter` permette di filtrare
- `tap`

Filter

```
const clicks = fromEvent(document, 'click');
const clicksOnDivs = clicks.pipe(filter(ev => (<HTMLElement>ev.target).tagName === 'DIV'));
clicksOnDivs.subscribe(x => console.log(x));
```



Debounce, throttle, backpressure

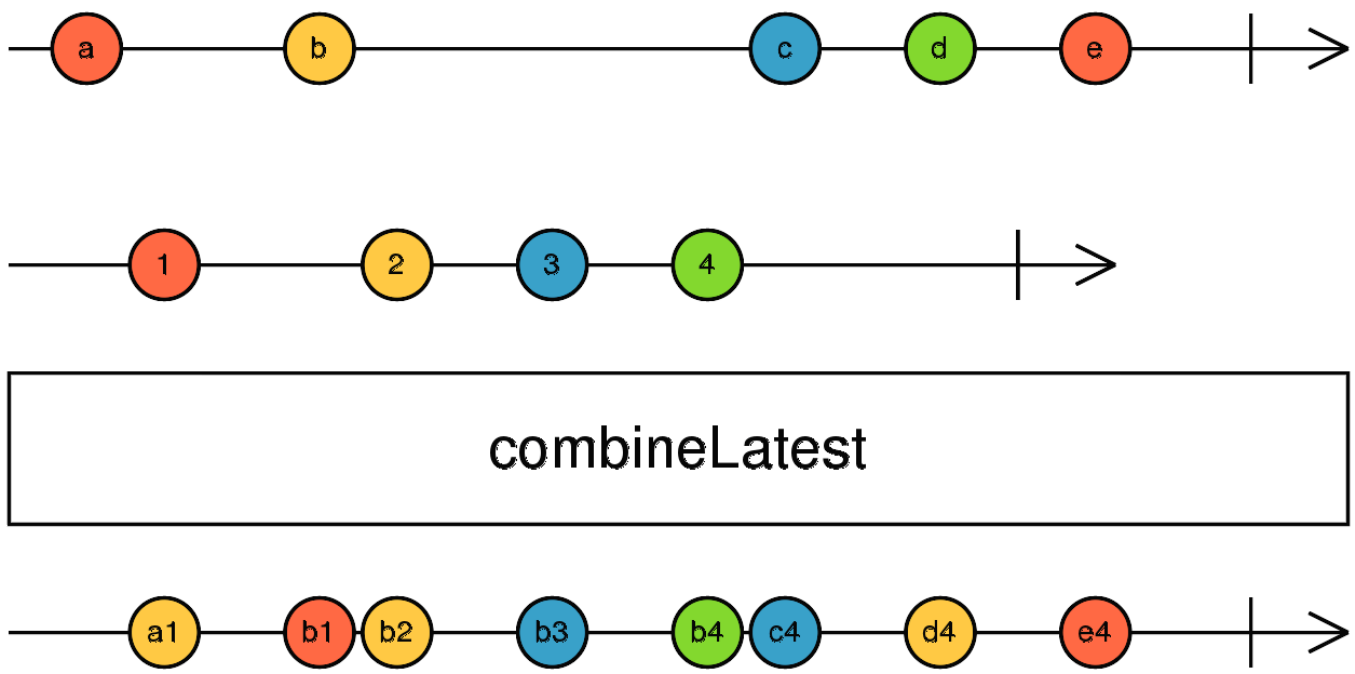
- `debounceTime` per input utente rumorosi
- `throttleTime` per eventi frequenti (scroll)
- Bilanciare velocità/accuratezza

Combinare sorgenti

- `combineLatest` (filtri + dati)
- `withLatestFrom` (usa stato al momento dell’evento)
- `forkJoin` (tutte complete → aggregazione)

CombineLatest

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

const firstTimer = timer(0, 1000); // emit 0, 1, 2... after every second, starting from now
const secondTimer = timer(500, 1000); // emit 0, 1, 2... after every second, starting 0,5s from now
const combinedTimers = combineLatest([firstTimer, secondTimer]);
```



Flattening: scegliere l’operatore giusto

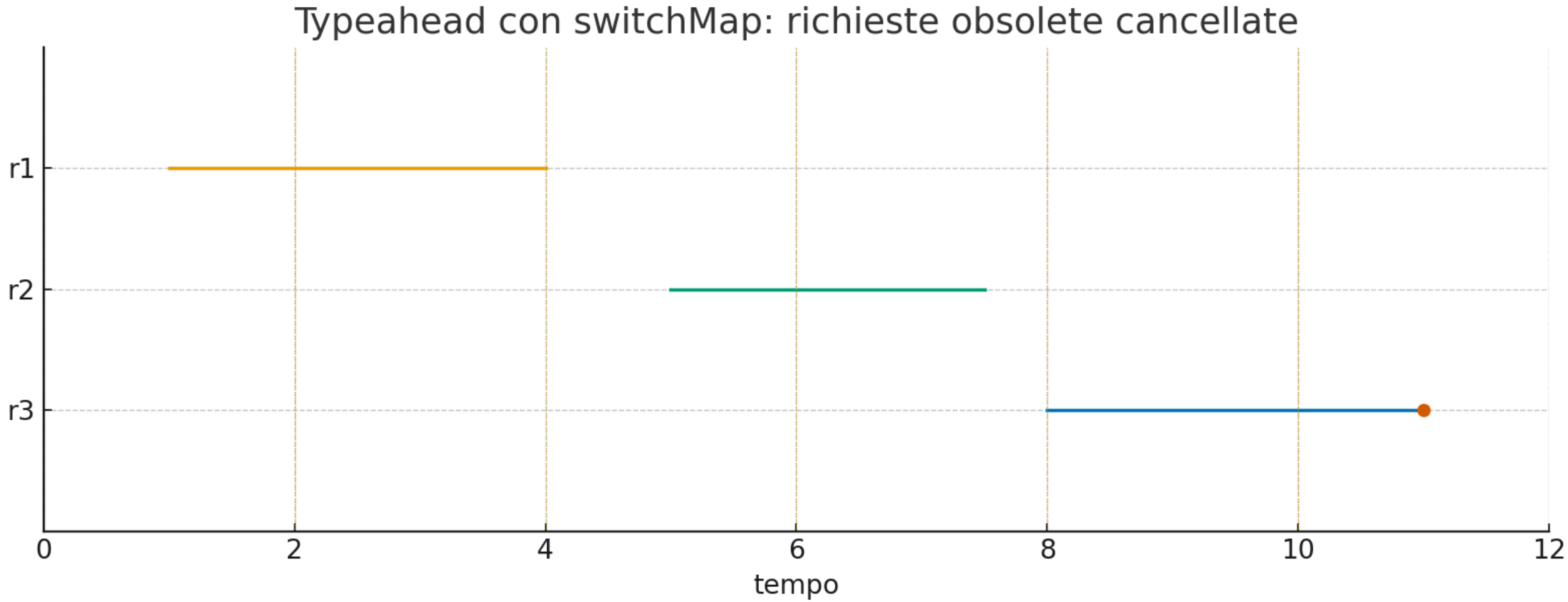
- `switchMap` : tieni solo l’ultimo (ricerca live, annulla i precedenti)
- `concatMap` : in **ordine** uno dopo l’altro (code di salvataggio)
- `mergeMap` : in **parallelo** (collezioni indipendenti)
- `exhaustMap` : ignora i nuovi finché c’è un lavoro in corso (evita doppio submit)

```
// TS
results$ = term$.pipe(
  debounceTime(250),
  distinctUntilChanged(),
  switchMap(q => this.svc.search(q))
);
```

Marbles — `switchMap`

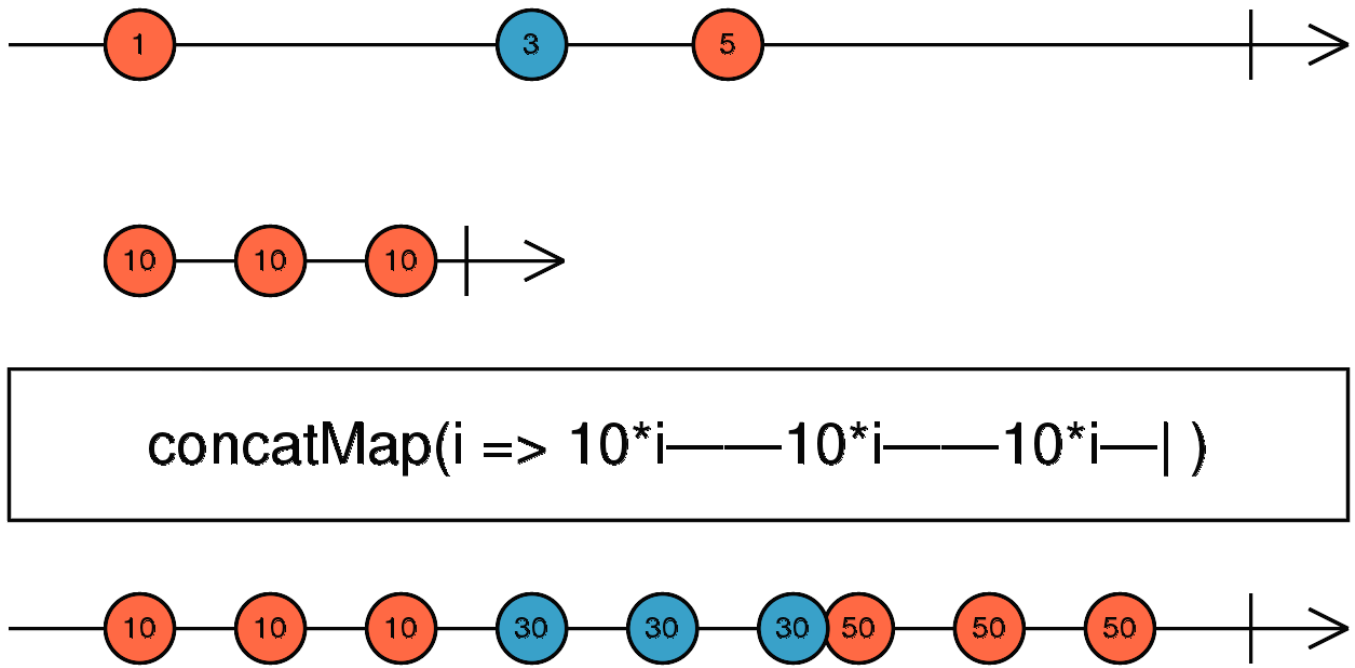
```
input:      --a---b-----c----|
richieste:    -r1----|   -r2----|   -r3----|
output:     -----r1X----r2X------r3----|
```

Legenda: `X` = cancellazione flusso precedente.


Typeahead con switchMap: richieste obsolete cancellate
r1
r2
r3
0
2
4
6
8
10
12
tempo

Esempio `concatMap`

```
saves$ = saveClicks$.pipe(
  concatMap(() => this.svc.saveForm())
);
```



Marbles — flattening a confronto

```
clicks:       --c---c-c----c---|
switchMap:    --s1X--s2X-s3X--s4-|
concatMap:    --s1----s2--s3----s4|
mergeMap:     --s1--s2s3--s4--|
exhaustMap:   --s1-----------(ignora)---s4|
```

Scegli in base a cancellazione, ordine o parallelismo richiesti.

Gestione degli errori

Un errore è **stato**, non un'eccezione da zittire. Mappiamo a un VM e, se ha senso, applichiamo **retry** solo per errori **transienti** (rete). Per cleanup grafico (spinner) usiamo `finalize`.

```
// TS
data$ = this.http.get(...).pipe(
  retry(1), // rete poco stabile
  catchError(err => of([])), // ritorna un array vuoto gestito
  finalize(() => this.loading=false)
);
```

Hot stream: quando usare i Subject

Usa `Subject/BehaviorSubject/ReplaySubject` **solo** per orchestrare input **locali** (click, pagina corrente), non come “variabili globali reattive”. Preferisci **derive** dagli stream quando possibile.

```
// TS
page$ = new BehaviorSubject(1);
paged$ = combineLatest([items$, page$]).pipe(
  map(([xs,p]) => xs.slice((p-1)*pageSize, p*pageSize))
);
```

Condivisione e cache: `shareReplay`

Se una sorgente è costosa (HTTP) e vuoi **riusare** il risultato tra più subscribe, usa `shareReplay({ bufferSize:1, refCount:true })`. Attenzione: senza `refCount` potresti mantenere la sorgente attiva anche senza subscriber.

```
products$ = this.http.get<Product[]>('/api/products').pipe(
  shareReplay({ bufferSize:1, refCount:true })
);
```

“Backpressure” in UI: non tutto va processato subito

Per input rumorosi (**scroll**, **resize**) applica una strategia:

- `throttleTime` : primo subito, poi silenzio per X ms
- `auditTime` : ultimo allo scadere della finestra
- `debounceTime` : emetti solo quando l’utente si ferma

```
// TS ≤12 – scroll “economico”
scroll$ = fromEvent(window,'scroll').pipe(auditTime(100));
```

Anti-pattern da evitare (e perché fanno male)

- **Subscribe annidati** → usa flattening (`switchMap` & co.)
- **Side-effect in `map`** → mettili in `tap`
- **Mancato unsubscribe** → `AsyncPipe` o `takeUntilDestroyed`
- **Subject dappertutto** → deriva dagli Observable esistenti
- **Pipe impure per "far funzionare"** → preferisci immutabilità/stream

Checklist pratica (prima di committare)

- Ho un `vm$` unico per la vista?
- Posso **cancellare** operazioni obsolete (search/submit)?
- Gli **errori** finiscono in uno stato leggibile?
- La lista usa `track` e gli stream non ricreano array mutando?
- Ho misurato il bisogno di **shareReplay** o posso evitare?

Recap generale

- Direttive per struttura/attributi (usa `@if/@for` + `track`)
- Pipe per presentazione (`async` per stream)
- Servizi con DI (`inject`) e scope consapevole
- **RxJS**: operatori chiave, combinazioni, errori, interop Angular