Angular 21

REAL-WORLD PATTERN

Debounce Search

RxJS vs Signals Showdown

`Traditional RxJS` `toSignal()` `toObservable()`

Which approach should YOU use? Let's find out!

@cb0401

Swipe →

Problem

The Search Problem

Users type fast. APIs are slow. Without debouncing, you'll fire hundreds of requests per second!

```
// BAD: Fires on EVERY keystroke
<input (input)="search($event)" />

search(event: Event) {
  const query = event.target.value;
  this.http.get(`/api?q=${query}`)  // 100+ requests!
}
```

`User types "angular"` → `7 API calls!`

WARNING: Without debouncing: Performance issues, rate limits, wasted bandwidth

Legacy

Traditional RxJS Approach

The classic way: Subject + pipe operators + manual subscription.

```
// Component
searchSubject = new Subject<string>();
results$: Observable<string[]>;

ngOnInit() {
  this.results$ = this.searchSubject.pipe(
    debounceTime(500),
    distinctUntilChanged(),
    filter(q => q.length > 2),
    switchMap(q => this.http.get(`/api?q=${q}`))
  );
}

// Template
<input (input)="searchSubject.next($event.target.value)" />
<div *ngFor="let r of results$ | async">{{ r }}</div>
```

- ✗ Verbose boilerplate code
- ✗ Async pipe or manual subscribe
- ✗ Unsubscribe management needed

Modern

Signal + toObservable()

The modern approach: Signal for state, convert to Observable for RxJS operators.

```
// State as Signal
searchQuery = signal('');

// Convert to Observable for RxJS operators
searchQuery$ = toObservable(this.searchQuery);

// Back to Signal for template
debouncedSearch = toSignal(
  this.searchQuery$.pipe(
    debounceTime(500),
    distinctUntilChanged(),
    filter(v => v.length > 3),
    map(v => v.toUpperCase())
  ),
  { initialValue: '' }
);
```

$$\begin{tikzcd}
\texttt{signal()} \arrow[r] & \texttt{toObservable()} \arrow[r] & \texttt{RxJS pipe} \arrow[r] & \boxed{\texttt{toSignal()}}
\end{tikzcd}$$

Clean Template Syntax

No async pipe needed! Just call the signal as a function.

```html
<!-- Input binds to signal -->
<input
  [value]="searchQuery()"
  (input)="searchQuery.set($event.target.value)"
/>

<!-- Display debounced result -->
<p>Searching for: {{ debouncedSearch() }}</p>
```

TIP: Signals auto-unsubscribe when the component is destroyed. No cleanup needed!

```typescript
// Bonus: React to changes with effect()
effect(() => {
  console.log('Search changed:', this.debouncedSearch());
});
```

Battle

Side-by-Side Comparison

Traditional RxJS

```
searchSubject = new Subject();
results$: Observable;

ngOnInit() {
  this.results$ = this.searchSubject
    .pipe(...)
}

ngOnDestroy() {
  // Cleanup needed!
}
```

Signal Approach

```
searchQuery = signal('');

debouncedSearch = toSignal(
  toObservable(this.searchQuery)
    .pipe(...)
);

// Auto cleanup!
```

~15
Lines of code

~8
Lines of code

When to Use What?

✓ Use Signal + toObservable()

When you need RxJS operators like debounce, but want Signal's simplicity in templates

✓ Use toSignal() for HTTP

Perfect for "fire and forget" streams like HTTP requests

✓ Use Pure RxJS

For complex event streams requiring WebSockets, retries, or multicasting

Rule of Thumb: Start with Signals, add RxJS only when needed

✓

Level Up Your Angular!

Signals + RxJS = The best of both worlds. Use them together for reactive, performant apps.

Less Boilerplate	Auto Cleanup	Better DX

Found this helpful?

Repost | Save | Follow