Angular Signals – End■to■End Guide (with Real■World Example)

> **Target stack:** Angular 17/18, Standalone APIs, Typescript 5+, RxJS 7+, Zonal & Zoneless CD

1) What are Signals?

Signals are **writable, synchronous state holders** with a getter/setter API that make **change detection** explicit and efficient. They replace ad■hoc `@Input()` mutation, many `BehaviorSubject` uses, and heavy RxJS chains for local UI state. Core pieces:

• `signal(initial: T)` – a reactive state container

• `computed(fn)` – derives new state from other signals

• `effect(fn)` – runs side effects when dependencies change

• Utilities: `untracked(fn)`, `peek()`, `batch(fn)`

• Interop: `toSignal(observable, opts)`, `toObservable(signal)`

• Component inputs/outputs modeled as signals via `input()`, `model()`, `output()` (Angular 17+)

Key properties

• *Pull■based*: values are read synchronously: `count()` returns `number`

• *Dependency tracking*: `computed/effect` re■run only when the signals they read change

• *Fine■grained updates*: updates propagate to only the consumers that read the signal

2) When to use Signals vs RxJS

Use case	Prefer Signals	Prefer RxJS
Local UI state (toggles, form fields, selections)	■	
Derived view model (filters, totals, sorting)	■	
Async data from HTTP/WebSocket	With `toSignal` for consumption	■ streams, operators, retries
Cross■component shared store	■ with a service (Signal Store)	■ NgRx/ComponentStore for complex
Complex event orchestration, cancellations		■

Rule of thumb: Use **Signals for local & derived view state**, use **RxJS for async/event coordination**, bridge with `toSignal`/`toObservable`.

3) Core API Quick Reference

```
import { signal, computed, effect, untracked, batch } from '@angular/core';

const qty = signal(1);
const price = signal(499);
const subtotal = computed(() => qty() * price());
```

```
const logFx = effect(() => {
  console.log('Subtotal changed:', subtotal());
});

batch(() => { qty.set(2); price.set(599); }); // one recompute
```

Writable helpers

```
qty.update(n => n + 1);
qty.mutate(v => { /* mutate object/array in place */ });
```

Interop

```
import { toSignal, toObservable } from '@angular/core/rxjs-interop';
```

4) Signal Inputs/Outputs/Models (Angular 17+)

```
import { Component, input, output, model } from '@angular/core';

@Component({
  selector: 'app-qty-stepper',
  standalone: true,
  template: `
    <button (click)="dec()">-</button>
    <span>{{ value() }}</span>
    <button (click)="inc()">+</button>
  `
})
export class QtyStepperComponent {
  // Two■way signal input
  value = model.required<number>();

  // Simple input signal with default
  step = input(1);

  // Output (still an EventEmitter under the hood)
  changed = output<number>();

  inc(){ this.value.update(v => v + this.step()); this.changed.emit(this.value()); }
  dec(){ this.value.update(v => Math.max(0, v - this.step())); this.changed.emit(this.value()); }
}
```

Usage

```
<app-qty-stepper [(value)]="qty"></app-qty-stepper>
```

5) Real■World End■to■End Example – **Product Catalog + Cart**

Scenario: ERP/E■commerce listing with search, filter, pagination, and a cart. We'll combine HTTP (RxJS) with local view■state (Signals) and demonstrate performance patterns.

5.1 Project Structure

```
src/app/
  services/catalog.service.ts     # HTTP + caching
  stores/cart.store.ts            # Signal store
```

```
  pages/catalog-page.ts           # Container component
  components/product-card.ts      # Presentational component
  components/qty-stepper.ts       # From section 4
```

5.2 CatalogService – HTTP + `toSignal`

```
// services/catalog.service.ts
import { Injectable, computed, signal } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { toSignal, toObservable } from '@angular/core/rxjs-interop';
import { map, shareReplay } from 'rxjs/operators';

@Injectable({ providedIn: 'root' })
export class CatalogService {
  private readonly api = '/api/products';

  // Query params as signals
  readonly q = signal('');
  readonly category = signal<string | null>(null);
  readonly page = signal(1);

  // Build an observable request whenever params change
  private readonly filters$ = toObservable(
    computed(() => ({ q: this.q(), category: this.category(), page: this.page() }))
  );

  // Replace mock$ with real HTTP when wiring:
  // const products$ = this.filters$.pipe(
  //   switchMap(params => this.http.get<ProductResponse>(this.api, { params: params as any })),
  //   shareReplay(1)
  // );

  private readonly mock$ = this.http.get<ProductResponse>(this.api).pipe(shareReplay(1));
  readonly data = toSignal(this.mock$, { initialValue: { items: [], total: 0 } });

  readonly items = computed(() => this.data().items);
  readonly total = computed(() => this.data().total);

  constructor(private http: HttpClient) {}
}
```

5.3 Cart Store – Pure Signals

```
// stores/cart.store.ts
import { Injectable, signal, computed, effect } from '@angular/core';

export interface CartLine { id: string; name: string; price: number; qty: number; }

@Injectable({ providedIn: 'root' })
export class CartStore {
  private readonly lines = signal<CartLine[]>([]);

  readonly subtotal = computed(() => this.lines().reduce((s, l) => s + l.price * l.qty, 0));
  readonly count    = computed(() => this.lines().reduce((s, l) => s + l.qty, 0));

  constructor(){
    // Persist to localStorage
    effect(() => {
      localStorage.setItem('cart', JSON.stringify(this.lines()));
    });

    // Load once
    const raw = localStorage.getItem('cart');
    if (raw) this.lines.set(JSON.parse(raw));
  }
```

```
  add(line: CartLine){
    this.lines.update(arr => {
      const i = arr.findIndex(x => x.id === line.id);
      if (i >= 0) { arr = arr.slice(); arr[i] = { ...arr[i], qty: arr[i].qty + line.qty }; return arr;
      return [...arr, line];
    });
  }

  setQty(id: string, qty: number){
    this.lines.update(arr => arr.map(l => l.id === id ? { ...l, qty } : l));
  }

  remove(id: string){ this.lines.update(arr => arr.filter(l => l.id !== id)); }
  clear(){ this.lines.set([]); }
}
```

5.4 Catalog Page – Container with Signals

```
// pages/catalog-page.ts
import { Component, inject } from '@angular/core';
import { CatalogService } from '../services/catalog.service';
import { CartStore } from '../stores/cart.store';
import { QtyStepperComponent } from '../components/qty-stepper';

@Component({
  selector: 'app-catalog-page',
  standalone: true,
  imports: [QtyStepperComponent],
  template: `
  <section>
    <input type="search" placeholder="Search" [value]="svc.q()" (input)="svc.q.set(($event.target as H
    <select [value]="svc.category() ?? ''" (change)="svc.category.set(($event.target as HTMLSelectEle
      <option value="">All</option>
      <option value="hotel">Hotel</option>
      <option value="dining">Dining</option>
      <option value="chauffeur">Chauffeur</option>
    </select>

    <div class="grid">
      <ng-container *ngFor="let p of svc.items()">
        <article class="card">
          <h3>{{ p.name }}</h3>
          <p>{{ p.price | currency:'INR' }}</p>
          <app-qty-stepper [(value)]="p.qty"></app-qty-stepper>
          <button (click)="add(p)">Add to Cart</button>
        </article>
      </ng-container>
    </div>

    <footer class="cart-bar">
      <strong>Items:</strong> {{ cart.count() }} | <strong>Subtotal:</strong> {{ cart.subtotal() | cur
    </footer>
  </section>
  `,
  styles: `
    .grid { display: grid; grid-template-columns: repeat(3, 1fr); gap: 12px; }
    .card { padding: 12px; border: 1px solid #ddd; border-radius: 8px; }
    .cart-bar { position: sticky; bottom: 0; background: #fff; padding: 8px; border-top: 1px solid #ee
  `
})
export class CatalogPageComponent {
  readonly svc = inject(CatalogService);
  readonly cart = inject(CartStore);

  add(p: any){
    this.cart.add({ id: p.id, name: p.name, price: p.price, qty: p.qty ?? 1 });
```

```
  }
}
```

5.5 Presentational `ProductCard` (Optional)

• Expose `product` as a **signal input** using `input()`

• Emit `add` using `output()`

```
import { Component, input, output } from '@angular/core';

@Component({
  selector: 'app-product-card',
  standalone: true,
  template: `
    <h3>{{ product().name }}</h3>
    <!-- ... -->
    <button (click)="add.emit(product())">Add</button>
  `
})
export class ProductCardComponent {
  product = input.required<any>();
  add = output<any>();
}
```

6) Zoneless Change Detection (Signals Friendly)

Signals shine with **zoneless CD** (Angular 17+).

• Bootstrap with `provideExperimentalZonelessChangeDetection()`

• Use signals/effects to notify views of changes explicitly

```
import { bootstrapApplication, provideExperimentalZonelessChangeDetection } from '@angular/platform-br

bootstrapApplication(AppComponent, {
  providers: [provideExperimentalZonelessChangeDetection()]
});
```

Guidelines

• Prefer signal inputs (`input()`) over classic `@Input`

• Wrap external callbacks (e.g., setTimeout, DOM events from outside Angular) to update signals

7) Forms with Signals

• Keep **FormGroup** for validation; mirror its value into a **signal view model**

• Or build lightweight forms as pure signals when validation is simple

```
const firstName = signal('');
const lastName = signal('');
const fullName = computed(() => `${firstName()} ${lastName()}`.trim());
```

Reactive Forms bridge

```
import { toSignal } from '@angular/core/rxjs-interop';
const formValue = toSignal(form.valueChanges, { initialValue: form.getRawValue() });
```

8) Interop Patterns

8.1 Observable → Signal

```
const data = toSignal(this.http.get<Data>('/x'), { initialValue: { ... } });
```

8.2 Signal → Observable

```
const filters$ = toObservable(computed(() => ({ q: this.q(), page: this.page() })));
```

8.3 Debounced Search

```
const search$ = toObservable(this.q).pipe(
  debounceTime(250), distinctUntilChanged(), switchMap(q => this.http.get('/search', { params: { q } }
);
this.results = toSignal(search$, { initialValue: [] });
```

9) Performance Playbook

• **Batch updates** with `batch(() => { ... })`

• Avoid reading the same signal repeatedly in templates; cache with `const v = sig()` in TS when doing heavy work

• Use `computed` for derived values instead of recalculating in the template

• Use **On■Push** (default in standalone) + signals or zoneless for minimal checks

• Prefer immutable updates (`update`/spread) for list changes; use `mutate` when safe for big arrays

• Split big components into smaller presentational units with **signal inputs**

10) Anti■Patterns & Gotchas

• **Don’t** perform side■effects in `computed`; use `effect`

• **Don’t** cause feedback loops: effects that update signals they read → use `untracked` inside effect when writing

• **Don’t** overuse `mutate` for nested structures; it can hide changes. Use `update` with a new reference for clarity

• **Beware** async re■entrancy; wrap multiple sequential updates in `batch`

```
const fx = effect(() => {
  const v = a();
  untracked(() => { b.set(v + 1); }); // avoid loop if b influences a
});
```

11) Testing Signals

```
import { signal, computed, effect } from '@angular/core';

describe('subtotal', () => {
  it('recomputes on deps change', () => {
    const qty = signal(1), price = signal(100);
    const subtotal = computed(() => qty() * price());
```

```
    expect(subtotal()).toBe(100);
    qty.set(2);
    expect(subtotal()).toBe(200);
  });
});
```

Component tests: read/write `model()` and assert DOM updates synchronously.

12) Migration Guide (BehaviorSubject → Signal)

1. Replace local `BehaviorSubject` with `signal(initial)`
2. Replace `.next(v)` with `.set(v)` / `.update(fn)`
3. Replace `combineLatest/map` view models with `computed`
4. Bridge remaining streams with `toSignal`/`toObservable`
5. Convert `@Input()` to `input()`/`model()` and outputs to `output()`
6. Remove unnecessary `async` pipes in templates where signals are used directly

Example

```
// Before
vm$ = combineLatest([a$, b$]).pipe(map(([a,b]) => ({ total: a*b })));

// After
const a = signal(2), b = signal(3);
const vm = computed(() => ({ total: a()*b() }));
```

13) NgRx & ComponentStore Co■existence

• Keep NgRx for domain events, effects, persistence, entity adapters

• Use Signals for component view state and selectors

• **Pattern**: Select from store → `toSignal(select$)`; dispatch actions from `effect` when local signals change

```
const selectedId$ = this.store.select(selectCurrentId);
const selectedId = toSignal(selectedId$, { initialValue: null });
```

14) Accessibility & UX with Signals

• Drive **ARIA states** from signals (`aria-expanded="isOpen()"`)

• Derive **disabled/hidden** conditions with `computed`

• Animate list additions/removals by reading signals in animation triggers

15) Error Handling Patterns

```
const state = signal<'idle'|'loading'|'error'|'ready'>('idle');
const errorMsg = signal<string | null>(null);

async function load(){
  state.set('loading');
  try {
```

```
    const data = await firstValueFrom(this.http.get('/api'));
    this.data.set(data);
    state.set('ready');
  } catch (e:any) {
    errorMsg.set(e.message ?? 'Unknown error');
    state.set('error');
  }
}
```

16) Security & SSR

• Signals are **framework■level only**; do not serialize sensitive data

• Works with **SSR/SSG**; treat signals as request■scoped in server render paths

17) Cheat■Sheet

• `signal(v)`, `set`, `update`, `mutate`, `peek`

• `computed(() => ...)` – pure, idempotent

• `effect(() => ...)` – for logging, DOM, network, store dispatch

• `input()`, `model()`, `output()` – component I/O as signals/events

• `toSignal(obs, { initialValue })`, `toObservable(sig)` – bridge

• `batch(() => { ... })`, `untracked(() => { ... })` – control recomputation

18) Appendix – Reusable Helpers

```
export function selectionSignal<T>(init: T[] = []){
  const list = signal<T[]>(init);
  const has = (x: T) => list().includes(x);
  const toggle = (x: T) => list.update(arr => has(x) ? arr.filter(i => i!==x) : [...arr, x]);
  return { list, has, toggle } as const;
}


export function paginationSignal(pageSize = 20){
  const page = signal(1);
  const next = () => page.update(p => p + 1);
  const prev = () => page.update(p => Math.max(1, p - 1));
  return { page, next, prev } as const;
}
```

19) Copy■Paste Checklist for New Screens

1. Define local state with `signal()`

2. Derive UI booleans/totals with `computed()`

3. Side■effects only in `effect()`

4. Inputs via `input()`/`model()`; outputs via `output()`

5. Async: keep RxJS in services; expose as `toSignal()` to components

6. Consider zoneless CD for performance■critical surfaces

7. Unit test `computed` and critical `effect`s

End

This document gives you a production■ready starting point to wire Signals across ERP modules (Hotel/Dining/Chauffeur/etc.), while co■existing with NgRx/Module Federation. Replace the mock HTTP call in `CatalogService` with your real endpoints and extend the cart store with taxes/discounts using `computed`.