Laboratorio Angular 18

Resumen técnico

- **Zoneless change detection (experimental en v18)**: Angular permite ejecutar aplicaciones sin `zone.js`. En zoneless el framework espera que los componentes (o Signals) notifiquen los cambios, lo que reduce ciclos innecesarios y tamaño de bundle.

- **Signals / `computed` / `effect`**: señales reactivas integradas en Angular: `signal()` para estado, `computed()` para valores derivados con tracking automático, y `effect()` para side-effects (se ejecutan al menos una vez y se re-ejecutan cuando sus dependencias cambian). Estas primitives son la base del modelo zoneless.

- **Control Flow (`@if`, `@for`, `@switch`)**: nueva sintaxis integrada en templates (ya presente desde v17 y estable en v18) que sustituye/ mejora a `*ngIf`/`*ngFor`: más cercana a JS, con mejor type narrowing y optimizaciones de render. `@for` exige `track` para forzar identidad.

- **Deferrable views / `@defer`**: permite marcar secciones del template para rendereado diferido (ideal para componentes pesados en páginas largas); funciona de forma declarativa en templates y ayuda a mejorar LCP.

Laboratorio paso a paso (hands-on)

Objetivo: crear una pequeña app de ejemplo con *standalone components*, Signals, zoneless bootstrap, control flow `@if`/`@for`, y `@defer` para diferir una vista pesada.

> Supuestos: tienes Node y Angular CLI ya instalados; tu entorno apunta a Angular 18 (si usas otra versión, actualiza @angular/* a 18.x).

1) Crear proyecto

```
ng new angular18-lab --standalone --routing=false --style=scss
cd angular18-lab
# (opcional) asegurarte de que @angular/* está en v18
npm install @angular/core@^18 @angular/cli@^18 @angular/compiler@^18
```

2) Estructura del ejercicio

- `src/main.ts` — bootstrap zoneless (experimental)
- `src/app/app.component.ts` — componente raíz **standalone**
- `src/app/services/state.service.ts` — ejemplo de servicio con Signals
- `src/app/components/list.component.ts` — muestra `@for`, `@if`
- `src/app/components/heavy.component.ts` — componente "pesado" (simulado) que cargaremos con `@defer`

3) `main.ts` — bootstrap zoneless (habilitar experimental zoneless)

Técnicamente en Angular 18 existen helpers para zoneless. En tu `main.ts` usa `bootstrapApplication` y el provider experimental (nombre exacto puede variar en parches menores; la guía oficial describe la forma de habilitar zoneless). Ejemplo:

```
// src/main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { provideRouter } from '@angular/router';
import { provideExperimentalZonelessChangeDetection } from '@angular/core'; // API experimental

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter([]),
    provideExperimentalZonelessChangeDetection() // activa zoneless (experimental)
  ]
}).catch(err => console.error(err));
```

Nota técnica: si quieres medir efecto, compila con y sin `zone.js` en `angular.json` (remueve import `'zone.js'` en polyfills) y compara bundles / inicialización.

4) Servicio de estado con Signals

Creamos un servicio *standalone* (no injection token complejo) que expone señales y `computed`:

```
// src/app/services/state.service.ts
import { Injectable, signal, computed } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class StateService {
  // estado básico
  items = signal<Array<{ id: number; name: string }>>([
    { id: 1, name: 'Alpha' },
    { id: 2, name: 'Bravo' },
    { id: 3, name: 'Charlie' }
  ]);

  // filtro como signal
  filter = signal<string>('');

  // computed: lista filtrada
  filtered = computed(() => {
    const f = this.filter().trim().toLowerCase();
    const arr = this.items();
    return f ? arr.filter(i => i.name.toLowerCase().includes(f)) : arr;
  });

  // mutators
  add(name: string) {
    const arr = this.items();
    const id = arr.length ? Math.max(...arr.map(i => i.id)) + 1 : 1;
    this.items.set([...arr, { id, name }]);
  }

  remove(id: number) {
    this.items.set(this.items().filter(i => i.id !== id));
  }

  setFilter(value: string) {
    this.filter.set(value);
  }
}
```

Usar `computed` en el servicio concentra la lógica derivada en un único lugar, y permite que componentes lean `filtered()` sin subscribirse explícitamente — mejora claridad y soporta zoneless.

5) Componente de lista: `@for` y `@if` en templates

Usamos componentes standalone y la nueva sintaxis `@for` / `@if`. Observa `track item.id` (mandatory in @for).

```
// src/app/components/list.component.ts
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { StateService } from '../services/state.service';

@Component({
  selector: 'app-list',
  standalone: true,
  imports: [CommonModule],
  template: `
    <div class="controls">
      <input placeholder="filter..." (input)="state.setFilter($any($event.target).value)" />
      <button (click)="state.add('New-' + Math.floor(Math.random()*1000))">Add</button>
    </div>

    <!-- @if control flow -->
    @if (state.filtered().length === 0) {
      <p>No items — try adding some.</p>
    } @else {
      <ul>
        <!-- @for with mandatory track -->
        @for (const item of state.filtered(); track item.id) {
          <li>
            <strong>{{ item.name }}</strong>
            <small>#{{ item.id }}</small>
            <button (click)="state.remove(item.id)">remove</button>
          </li>
        } @empty {
          <li>No items</li>
        }
      </ul>
    }

  `
})
export class ListComponent {
  constructor(public state: StateService) {}
}
```

Puntos técnicos:

- `@for` requiere `track` para una identificación clara y DOM minimal updates; evita operaciones innecesarias cuando listas cambian. `@if` provee mejor type narrowing y sintaxis más legible que `*ngIf`.

6) Componente pesado y @defer

Simulamos un componente costoso que tarda en cargar (por ejemplo, un gráfico grande). Lo marcamos para carga diferida desde el template con `@defer`.

```
// src/app/components/heavy.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-heavy',
  standalone: true,
  template: `
    <div class="heavy">
      <h3>Heavy component (simulated)</h3>
      <p>Imagina aquí un gráfico grande o librería externa.</p>
      <p>Rendered at: {{ now }}</p>
    </div>
  `
})
export class HeavyComponent {
  now = new Date().toISOString();
}
```

En el `AppComponent` lo envolvemos con `@defer`:

```
// src/app/app.component.ts
import { Component } from '@angular/core';
import { ListComponent } from './components/list.component';
import { HeavyComponent } from './components/heavy.component';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, ListComponent],
  template: `
    <h1>Angular 18 Lab - Zoneless + Signals + Control Flow + Defer</h1>
```

```
    <app-list></app-list>

    <!-- Deferrable view -->
    @defer {
      <app-heavy></app-heavy>
    } @placeholder {
      <div class="placeholder">Loading heavy component…</div>
    } @loading {
      <div class="skeleton">…still loading</div>
    }
  `
})
export class AppComponent {}
```

`@defer` permite que la vista pesada se cargue más tarde (por scroll o timeout según configuración) reduciendo el bundle inicial y mejorando métricas de carga (LCP).

7) Efectos y sincronización con DOM / APIs externas

Un `effect()` en un componente o servicio sirve para mantener sincronía con el mundo no-reactivo (ejemplo: logging o integración con librerías imperativas). Ejemplo en el `ListComponent`:

```
// añadir en ListComponent (constructor)
import { effect } from '@angular/core';

constructor(public state: StateService) {
  // efecto para log / analytics (se ejecuta inicialmente y cuando cambia filtered)
  effect(() => {
    const count = state.filtered().length;
    console.log(`[effect] filtered count = ${count}`);
    // aquí podrías llamar a una API de analytics de manera controlada
  });
}
```

Los `effect()` siempre se ejecutan asíncronamente durante el ciclo de change detection; si haces side effects costosos, limpia/optimiza dentro del `effect` (usa `onCleanup` si necesitas liberar recursos).

8) Comparativa práctica: `@if` vs `*ngIf` vs `[hidden]`

- `@if` / `@for`: nueva sintaxis optimizada por Angular; mejor type narrowing y menos imports. Recomendado para nuevo código.

- `*ngIf` sigue soportado pero es un *structural directive* que depende de CommonModule; con `@if` evitas importar `CommonModule`.

- `[hidden]` (CSS `display:none`) **no** remueve nodes del DOM — útil cuando necesitas mantener estado DOM o evitar re-render costoso; pero **no** reduce el trabajo del parser/paint inicial. Para optimizar LCP y reduce JS, preferir `@defer` o `@if` para evitar render de subtrees pesados. (Resumen práctico: `[hidden]` = ocultar visualmente; `@if`/`*ngIf` = no renderizar; `@defer` = mantener cargado diferido).

9) Pruebas / verificación

- `ng serve` y abrir la app. Revisa consola para ver logs del `effect`.

- Construye en prod: `ng build --prod` y compara bundle size con/ sin `zone.js` importado.

- Usa Lighthouse para medir LCP / TBT con y sin `@defer` en páginas largas.

Consejos

- **Zoneless es experimental**: probar en nuevas aplicaciones o en feature branches; mide siempre. Algunas librerías externas que esperan zone.js pueden no comportarse igual → test de integración obligatorio.

- **Señales vs RxJS**: Signals son fantásticas para estado local y derivaciones; RxJS sigue siendo útil para streams complejos y operadores que Signals no cubren de forma natural. Considera patrones híbridos (servicio Signal que exponga `subscribe()` para compatibilidad si se necesita).

- **`effect()` cuidado**: efectos re-ejecutarán al cambiar dependencias; evitar side-effects costosos en cada ejecución sin controles. Usa `onCleanup` para liberar timers/intervalos.

- **@for require track**: si no usas tracking correcto, podrías incurrir en re-renders costosos; `track item.id` es la práctica recomendada para colecciones dinámicas.

1. ¿Qué es `zone.js` y por qué se puede eliminar?

En versiones anteriores de Angular (hasta la 17), `zone.js` era **obligatorio**:
Interceptaba eventos del navegador (clicks, XHR, timers…) y avisaba a Angular para ejecutar *change detection* automáticamente.

Esto es cómodo, pero tiene un coste:

- Más listeners globales (intercepta casi todo el runtime del navegador).
- Change detection más frecuente (a veces se ejecuta aunque no haya cambios reales).
- Mayor peso en bundle (~30 KB minificados).

En Angular 18, el framework introduce **zoneless change detection**, que **elimina esa dependencia** y se apoya en **Signals** para saber cuándo algo ha cambiado.

2. Proyecto base

Supongamos que tienes tu proyecto creado así:

ng new angular18-lab --standalone --routing=false --style=scss
cd angular18-lab

3. Versión A — Compilar con `zone.js` (modo clásico)

Este es el comportamiento por defecto.

Abre `src/polyfills.ts` (si no existe, Angular lo resuelve internamente) y asegúrate de tener:

```
import 'zone.js'; // Included with Angular CLI.
```

En `main.ts` usa el bootstrap clásico (sin el provider zoneless):

```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent)
  .catch(err => console.error(err));
```

Compila:

```
ng build --configuration production
```

O en desarrollo:

```
ng serve
```

Este build incluye `zone.js` y la aplicación funcionará con detección de cambios tradicional.

4. Versión B — Compilar sin `zone.js` (modo ZONELESS)

Aquí eliminaremos por completo `zone.js` y activaremos la nueva detección basada en *signals*.

1. **Quita la importación de `zone.js`**:
    - Abre `src/polyfills.ts` o verifica en `angular.json` que no se incluya.
    - Si ves la línea `"zone.js"`, elimínala.

Agrega el provider experimental en `main.ts`:

```
 import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { provideExperimentalZonelessChangeDetection } from '@angular/core';

bootstrapApplication(AppComponent, {
  providers: [provideExperimentalZonelessChangeDetection()]
}).catch(err => console.error(err));
```

Compila y ejecuta:

```
 ng build --configuration production
ng serve
```

Verifica que Angular ya no carga zone.js:

- Abre las DevTools → pestaña *Network* → busca “zone.js”.
- Si no aparece, estás realmente en modo zoneless.

También puedes poner en consola:

```
 console.log((window as any).Zone); // undefined → Zoneless OK
```

-

5. Comparación de rendimiento

Aquí tienes un mini-plan de comparación:

Métrica	Con Zone.js	Sin Zone.js	Observaciones
Bundle Size	Mayor (~30 KB más)	Menor	Mide con `dist/`
Change Detection	Automática y global	Manual (vía Signals)	Menos CPU
Memory footprint	+Listeners y patches	Limpio	Menos overhead
Lighthouse / LCP / TBT	5–10 % más lento (según caso)	Más rápido	Sobre todo en apps con muchos listeners

Puedes usar Lighthouse o Web Vitals:

npx lighthouse http://localhost:4200 --view

O herramientas como `ng profiler` o las DevTools de Angular (v18+ soportan zoneless profiling).

6. A tener en cuenta

- **No todo funciona zoneless aún**:
  Si usas librerías que dependen de Zone.js (por ejemplo, `ngx-bootstrap`, `ng-zorro`, o algunos wrappers de librerías JS), podrían fallar.
  Puedes activarlo solo en partes nuevas de la app mientras comparas.

- **Signals son clave**:
  Si quitas `zone.js`, Angular **no sabe cuándo algo cambió**, a menos que uses `signal()`, `computed()` o `effect()`.

- **Eventos de DOM siguen funcionando**, pero ya no disparan detección automática a menos que actualicen una signal.

Automatizar la comparación

Puedes mantener ambos modos en ramas distintas para comparar:

```
git checkout -b with-zone
# mantiene zone.js

git checkout -b zoneless
# quita zone.js y añade provideExperimentalZonelessChangeDetection
```

Después construyes y comparas tamaños:

```
du -sh dist/*
```

Y ejecutas Lighthouse en ambos builds.

Resumen

Aspecto	Con zone.js	Zoneless (Angular 18)
Detección de cambios	Automática, global	Basada en Signals
Rendimiento CPU	Más alto	Más eficiente
Tamaño del bundle	+30 KB aprox	Menor
Compatibilidad	Total	Experimental (18.x)
Ideal para	Apps legacy	Apps nuevas, SPA reactivas

GLOSARIO TÉCNICO — Angular, Rendimiento y Web

Angular

Definición técnica: Framework front-end desarrollado por Google para construir aplicaciones web de una sola página (SPA) mediante componentes y un sistema de plantillas declarativas.
Explicación sencilla: Es una herramienta que nos permite crear páginas web dinámicas con estructura modular y reactividad.

Zone.js

Definición técnica: Librería que intercepta las operaciones asíncronas del navegador (clicks, temporizadores, peticiones HTTP, etc.) para avisar a Angular cuándo debe ejecutar *change detection*.
Explicación sencilla: Es como un "vigilante" que detecta cuando algo cambia en la página para que Angular la actualice automáticamente.
En Angular 18: Se puede eliminar si usamos Signals (modo *zoneless*), lo que mejora rendimiento.

Zoneless (modo sin Zone.js)

Definición técnica: Modo de Angular donde la detección de cambios se basa exclusivamente en *signals* y no en interceptar eventos globales.
Explicación sencilla: Angular solo actualiza lo que realmente ha cambiado, sin estar "escuchando" todo el tiempo el navegador. Es más rápido y consume menos recursos.

Change Detection (Detección de cambios)

Definición técnica: Mecanismo interno que sincroniza el estado de los componentes con el DOM.
Explicación sencilla: Es el proceso por el que Angular revisa si algo en los datos ha cambiado y actualiza la vista en pantalla.

Signal

Definición técnica: Función reactiva de Angular que almacena un valor y notifica automáticamente a los componentes cuando este cambia.
Explicación sencilla: Es una variable "inteligente" que avisa a Angular para que actualice la vista cuando su valor cambia.

Computed

Definición técnica: Signal derivado que calcula un valor basado en otros signals. Se recalcula automáticamente cuando sus dependencias cambian.
Explicación sencilla: Es como una fórmula en Excel: si cambian los datos, el resultado se actualiza solo.

Effect

Definición técnica: Bloque de código que se ejecuta como respuesta a cambios en uno o varios signals (por ejemplo, para registrar logs o actualizar una API).
Explicación sencilla: Es una “reacción automática” a un cambio: cuando algo cambia, Angular ejecuta una acción.

Standalone Component

Definición técnica: Componente independiente introducido en Angular 14 que no necesita declararse en un módulo (`NgModule`).
Explicación sencilla: Un componente autosuficiente: se puede usar directamente sin necesidad de módulos intermedios.

Control Flow (`@if`, `@for`, `@switch`)

Definición técnica: Nueva sintaxis de Angular (v17+) que reemplaza a las directivas estructurales `*ngIf` y `*ngFor`, con mejor rendimiento y type checking.
Explicación sencilla: Son las nuevas formas de escribir condiciones y bucles en los templates, más parecidas a la sintaxis de JavaScript.

@defer / Carga diferida

Definición técnica: Directiva de Angular que retrasa la renderización o descarga de partes del template hasta que se cumplen ciertas condiciones (por ejemplo, scroll o interacción).
Explicación sencilla: Angular espera un poco antes de cargar partes pesadas de la página, así el usuario ve antes lo importante.

Bundle

Definición técnica: Archivo resultante del proceso de compilación de Angular que contiene todo el código JavaScript, CSS y recursos necesarios para ejecutar la aplicación.
Explicación sencilla: Es el “paquete” final que el navegador descarga para mostrar la web. Cuanto más pequeño, más rápido carga.

Lazy Loading (Carga perezosa)

Definición técnica: Estrategia que carga partes del código solo cuando se necesitan (por ejemplo, un módulo o componente).
Explicación sencilla: En vez de cargar toda la app desde el principio, se carga lo que el usuario necesita en ese momento.

LCP (Largest Contentful Paint)

Definición técnica: Métrica de rendimiento web que mide el tiempo que tarda en mostrarse el elemento más grande y visible del viewport.
Explicación sencilla: Cuánto tarda en aparecer el contenido principal de la página (por ejemplo, una imagen o un texto grande).
Objetivo ideal: Menos de 2,5 segundos.

TBT (Total Blocking Time)

Definición técnica: Tiempo total durante el cual el hilo principal del navegador está bloqueado impidiendo la interacción del usuario.
Explicación sencilla: Cuánto tiempo la página “se queda pensando” y no responde al hacer clic o desplazarte.

FCP (First Contentful Paint)

Definición técnica: Tiempo que tarda el navegador en mostrar el primer elemento de contenido (texto o imagen).
Explicación sencilla: Es el instante en que “empieza a verse algo” en la pantalla.

Lighthouse

Definición técnica: Herramienta de auditoría de rendimiento desarrollada por Google que mide velocidad, accesibilidad y SEO de una web.
Explicación sencilla: Un analizador automático que te dice si tu web es rápida, accesible y está bien optimizada.

Build / Compilación

Definición técnica: Proceso mediante el cual Angular transforma el código TypeScript y los templates en JavaScript optimizado para navegadores.
Explicación sencilla: Es como “empaquetar” y optimizar tu aplicación para que el navegador pueda entenderla y ejecutarla rápido.

SPA (Single Page Application)

Definición técnica: Aplicación web que carga una sola página HTML y actualiza dinámicamente el contenido mediante JavaScript sin recargar toda la página.
Explicación sencilla: Es una web que parece una aplicación: no se recarga entera al navegar.

DOM (Document Object Model)

Definición técnica: Estructura jerárquica que representa el contenido HTML de una página web.
Explicación sencilla: Es como un árbol que contiene todos los elementos visibles (botones, textos, imágenes…) del sitio web.

Reactividad

Definición técnica: Paradigma en el que los cambios en los datos se propagan automáticamente a la interfaz sin necesidad de código imperativo adicional.
Explicación sencilla: Cuando cambian los datos, la vista se actualiza sola. Es el principio detrás de Signals, RxJS o Vue.

RxJS (Reactive Extensions for JavaScript)

Definición técnica: Librería para programación reactiva basada en observables que Angular ha usado tradicionalmente para manejar flujos de datos asíncronos.
Explicación sencilla: Un sistema para escuchar y reaccionar a flujos de datos (como clicks, HTTP o timers).
En Angular 18: Coexiste con Signals; ambos pueden convivir según el caso.

Build Optimizer / Tree Shaking

Definición técnica: Proceso del compilador que elimina código no utilizado para reducir el tamaño final del bundle.
Explicación sencilla: Angular borra el código sobrante antes de generar el paquete final, para que pese menos.

Change Detection Cycle

Definición técnica: Secuencia de pasos en los que Angular revisa cada componente para determinar si debe actualizar su vista.
Explicación sencilla: Es el ciclo de “comprobación de cambios” que antes ejecutaba Zone.js y que ahora puede hacer Signals automáticamente.

Placeholder / Skeleton

Definición técnica: Contenido temporal mostrado mientras se carga otro componente o recurso.
Explicación sencilla: Es una especie de “marco vacío” o “hueso” que se muestra mientras el contenido real se está cargando.

Track (en `@for`)

Definición técnica: Expresión usada en `@for` para identificar de forma única los elementos de una lista y optimizar su renderizado.
Explicación sencilla: Sirve para que Angular sepa qué elemento de la lista cambió, sin tener que volver a dibujarlos todos.

Viewport

Definición técnica: Parte visible del documento web en la pantalla del usuario.
Explicación sencilla: Es la “ventana” que el usuario ve en su navegador en un momento dado.

Polyfill

Definición técnica: Fragmento de código que añade soporte en navegadores antiguos para características modernas de JavaScript o del DOM.
Explicación sencilla: Es un “parche” que permite usar funciones nuevas en navegadores que todavía no las entienden.
