GALAXY
TRAINING

Ing. ERICK ARÓSTEGUI CUNZA

Sesión 01

Angular en Acción: Reactividad, DI y Base Modular

Instructor

ERICK AROSTEGUI CUNZA

erick.arostegui.cunza@gmail.com

GALAXY TRAINING

ARQUITECTURA DE APLICACIONES

MULTI-STACK AVANZADO

Angular 21

“La arquitectura no es complicar el código, es hacerlo crecer sin romper”.

01 Cómo funciona Angular internamente

02 Directivas Modernas

03 Renderizado Diferido en Angular

04 Signals en Profundidad

05 Inyección de Dependencias Moderna en Angular

06 Estructura Base del Proyecto

07 Angular Material + Theming Base

01 - CÓMO FUNCIONA ANGULAR INTERNAMENTE

¿Qué hace Angular cuando arranca?

Hoy el comando correcto es:

ng dev

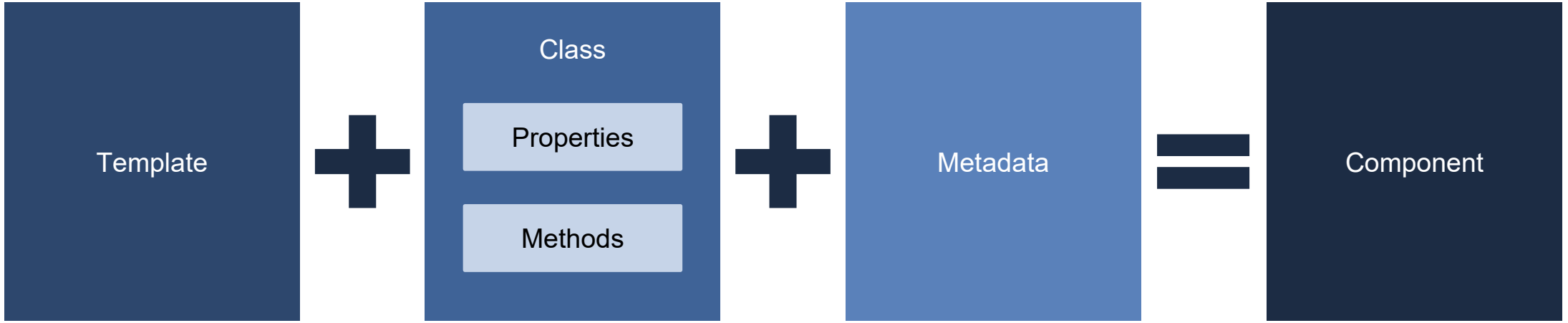
(Angular CLI moderno usa el nuevo builder con esbuild y servidor optimizado).

Cuando arrancamos el proyecto:

1. Angular compila con esbuild.
2. Genera el bundle optimizado.
3. Inicia el dev server.
4. Ejecuta main.ts.
5. Hace bootstrap con **bootstrapApplication**.

En Angular 21 ya no usamos **AppModule**.

¿Qué Es un Componente Realmente?



- View layout
- Usa HTML
- Incluye binding y directivas

- Código de soporte a la vista
- Usa TypeScript
- Propiedades: data
- Metodos: logica

- Información adicional para la ejecución de Angular
- Definido por decoradores

¿Qué Es un Componente Realmente?

app.component

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

@Component({
    selector: 'pm-root',
    template: `
    <div><h1>{{pageTitle}}</h1>
        <div>My First Component</div>
    </div>
    `
})
export class AppComponent {
 pageTitle: string = 'Acme Product Management';
}
```

Import

Metadata & Template

Class

Árbol de Componentes (Standalone First)

```
@Component({
  selector: 'app-home',
  imports: [
    MatButtonModule,
    MatCardModule,
    MatIconModule,
    MatToolbarModule,
    MatMenuModule,
    ThemeToggle,
    ConfigViewerButton,
    TokenInfoButton,
  ],
  templateUrl: './home.html',
  styleUrl: './home.scss',
})
export class Home {
  protected readonly security = inject(SecurityService);
  private readonly router = inject(Router);

  accessModule(module: string): void {
    this.router.navigate(['/library']);
  }
}
```

Angular 21 es standalone-first.
No necesitamos NgModules para arrancar.

Estructura mental:

AppComponent
↓
ShellComponent
↓
FeatureComponent

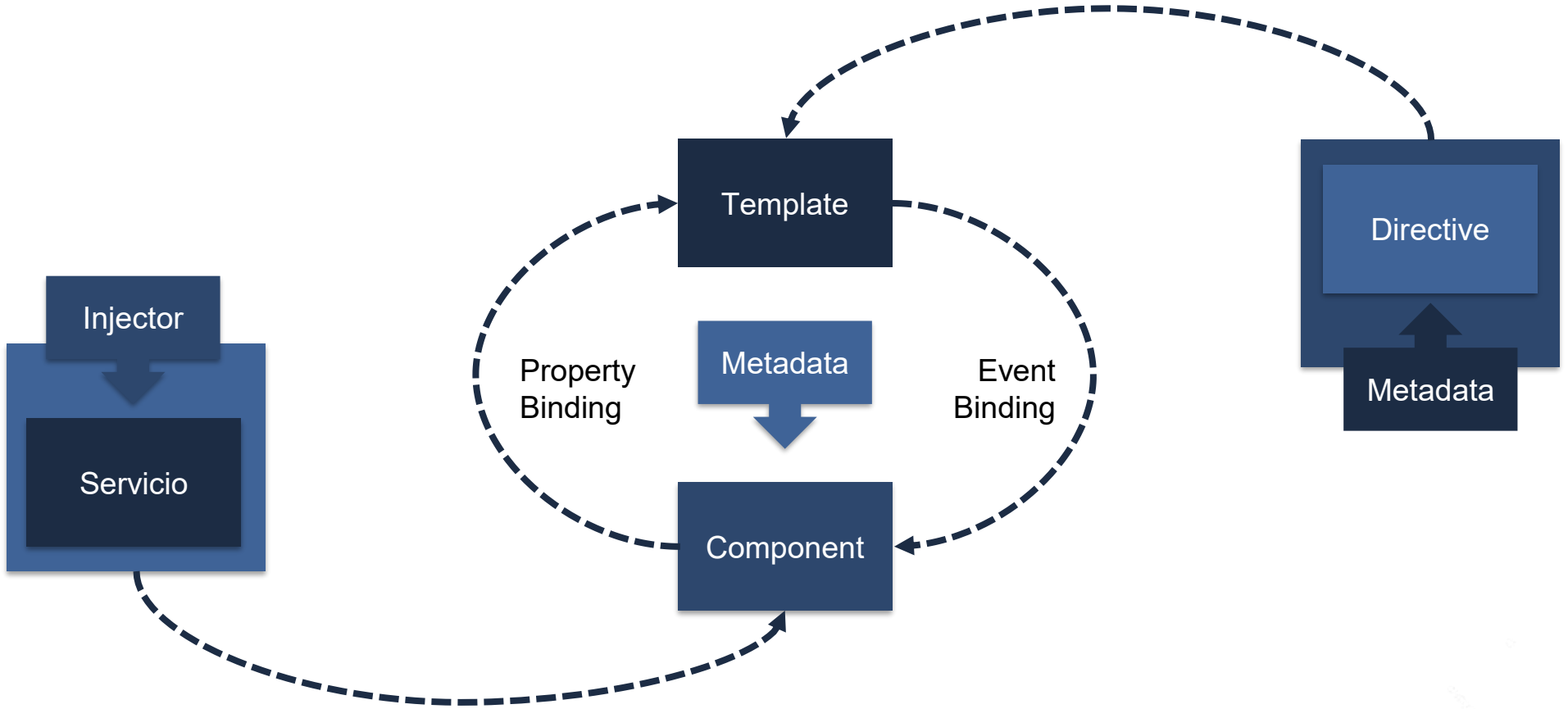
Cada componente:

- Es standalone.
- Declara sus imports.
- Tiene su propio contexto.

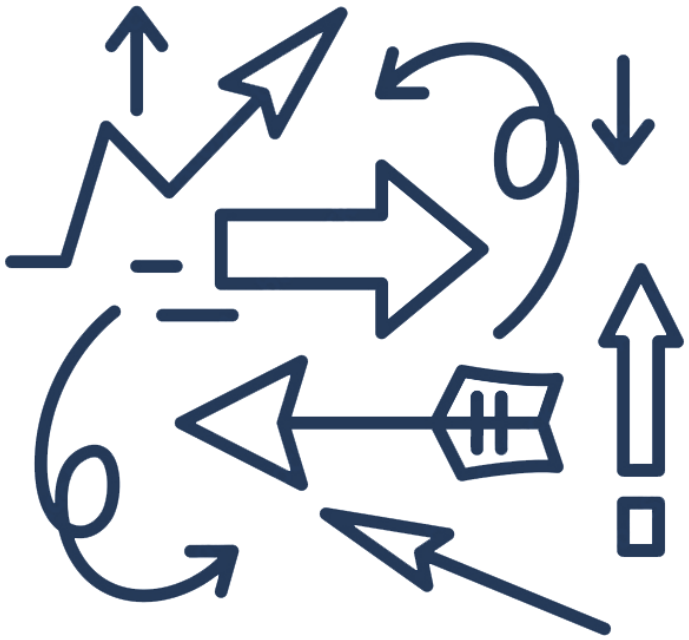
La aplicación es un árbol reactivo.

Cómo funciona Angular internamente

Esquema de ejecución de componentes



Change Detection en Angular



Angular actualiza la vista cuando:

- Hay eventos
- Cambian signals
- Hay peticiones HTTP
- Se ejecutan efectos

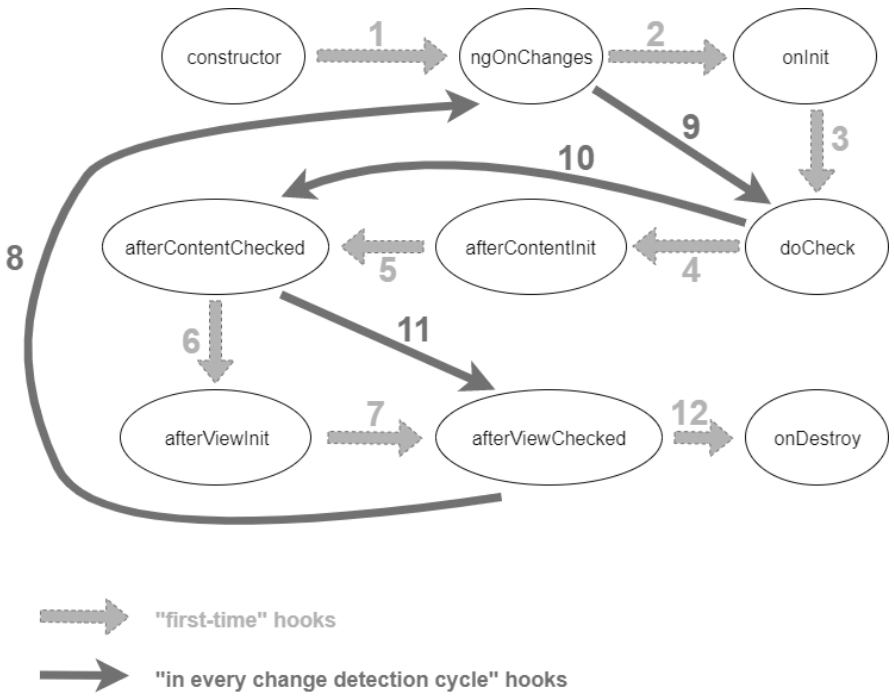
Con signals:

- Angular sabe exactamente qué cambió
- Reduce trabajo innecesario
- Hace actualizaciones más precisas

Ya no dependemos tanto de suscripciones manuales.

Cómo funciona Angular internamente

Ciclo de Vida Moderno



Momentos importantes:

- constructor
- ngOnInit
- ngOnDestroy

Pero ahora:

- Podemos usar inject()
- Podemos usar effect()
- Podemos evitar lógica innecesaria en hooks

```
private bo = inject(BreakpointObserver);
```

02 – DIRECTIVAS MODERNAS

¿Qué Son las Directivas?

Una directiva es una instrucción que le dice a Angular:

- Cómo modificar el DOM
- Cuándo renderizar algo
- Cómo reaccionar a cambios

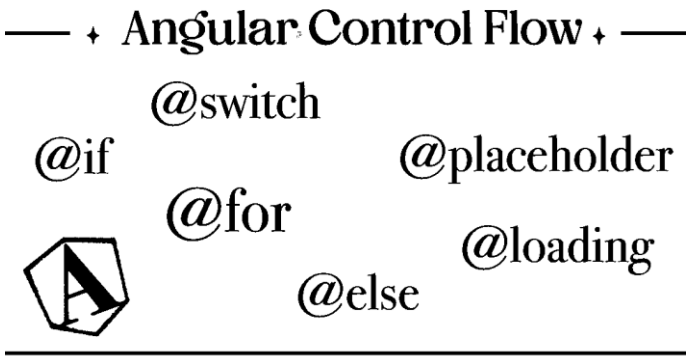
En Angular distinguimos:

1. Componentes (directivas con template)
2. Directivas estructurales (@if, @for, etc.)
3. Directivas de atributo (ngClass, ngStyle)

Las directivas hacen dinámico el template.

Directivas Modernas

Nuevo Control Flow (Reemplazo del Angular Clásico)



Antes:

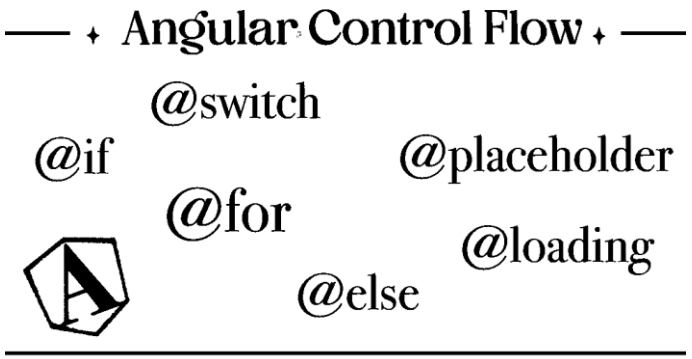
- *ngIf
- *ngFor
- [ngSwitch]

Ahora:

- @if
- @for
- @switch

Sintaxis más limpia, más cercana a JavaScript.

Directivas Estructurales (Angular Moderno)



Las directivas estructurales modifican la estructura del DOM.

En Angular usamos:

- @if
- @for
- @switch

Ejemplo moderno:

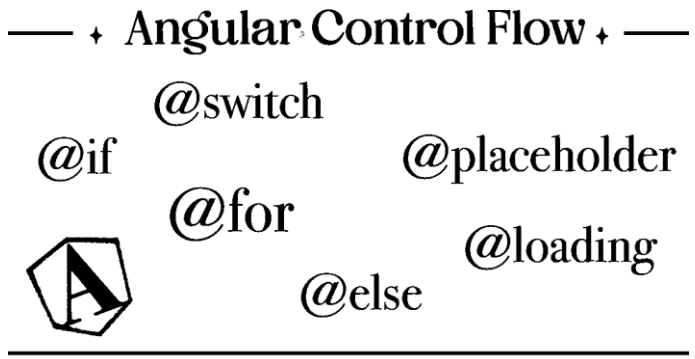
```
@if (author().isDeceased) {
<mat-icon matPrefix>church</mat-icon>
} @else {
<mat-icon matPrefix>sentiment_very_satisfied</mat-icon>
}
```

Ventajas:

- Sintaxis más clara.
- Mejor tipado.
- Más cercana a JavaScript.

Directivas Modernas

@for (Reemplazo de *ngFor)



Ejemplo moderno:

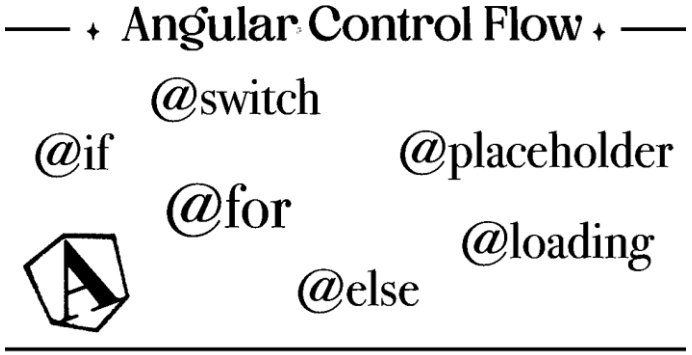
```
@for (author of authors(); track author.id) {
  <p>{{ author.name }}</p>
}
```

Ventajas:

- Mejor rendimiento con track.
- Sintaxis más limpia.
- Mejor integración con signals.

Angular moderno optimiza iteraciones automáticamente.

Directivas Modernas



@let

Permite declarar una variable en el template basada en una expresión.

Se actualiza automáticamente cuando cambia la expresión.

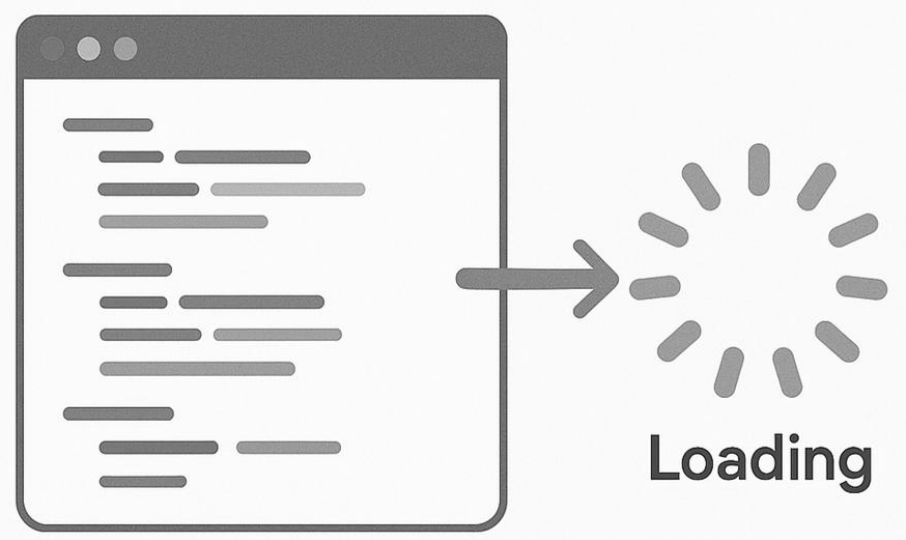
Evita repetir llamadas complejas en el template.

```
@let total = items().length;
<p>Total de elementos: {{ total }}</p>
```

```
@let activeUsers = users().filter(u => u.active);
<p>Activos: {{ activeUsers.length }}</p>
```

03 – RENDERIZADO DIFERIDO EN ANGULAR

¿Qué es @defer?



@defer permite **posponer la carga y renderizado de un bloque** hasta que se cumpla una condición.

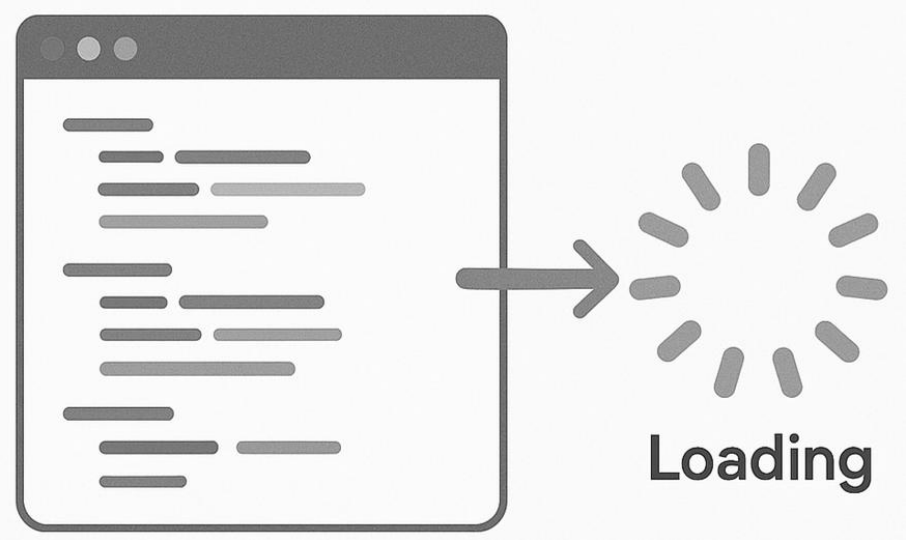
No solo oculta contenido.

Angular puede:
- Dividir el código en un chunk separado.
- No incluirlo en el bundle inicial.
- Descargarlo solo cuando sea necesario.

Es optimización real de performance.

Renderizado Diferido en Angular

Sintaxis Base



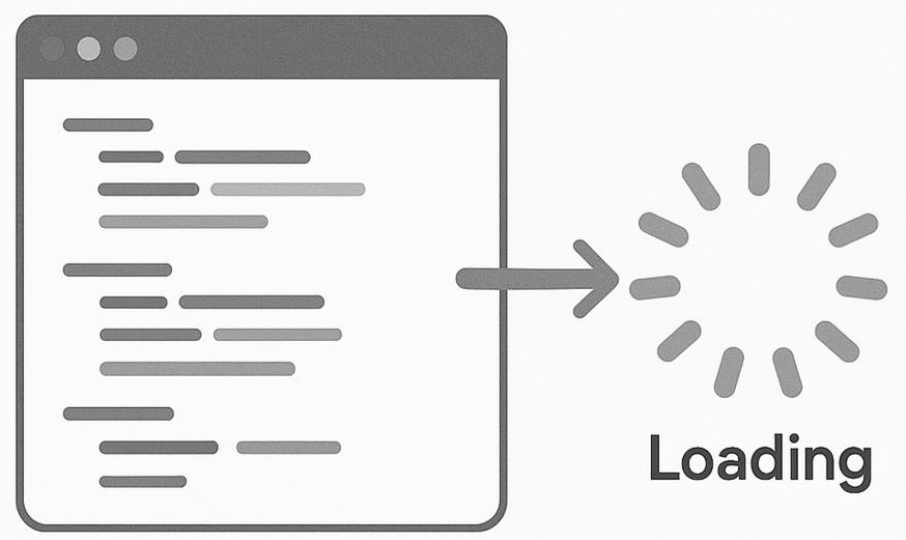
Ejemplo:

```
@defer {
    <heavy-component />
}
```

Angular:

1. Separa ese componente en un chunk.
2. No lo carga al inicio.
3. Lo descarga cuando se activa el trigger.

Triggers Disponibles



Puedes controlar cuándo se carga:

on viewport

Se activa cuando entra en pantalla.

```
@defer (on viewport) {
   <chart-component />
}
```

on interaction

Se activa cuando el usuario interactúa.

```
@defer (on interaction) {
    <details-panel />
}
```

on idle

Cuando el navegador está inactivo.

```
@defer (on idle) {
   <secondary-widget />
}
```

on timer

```
@defer (on timer(3000)) {
   <promo-banner />
}
```

Renderizado Diferido en Angular

Bloques Internos

```
@defer (on viewport) {
    <chart-component />
} @placeholder {
    <p>Cargando gráfico...</p>
} @loading {
    <spinner />
} @error {
    <p>Error al cargar</p>
}
```

@defer puede incluir:

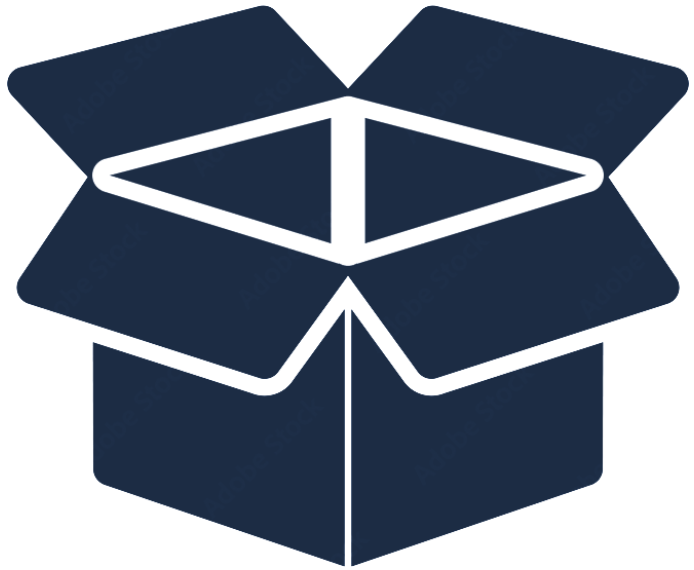
@placeholder
Antes de cargar.

@loading
Mientras descarga el chunk.

@error
Si falla la carga.

04 – SIGNALS EN PROFUNDIDAD

¿Qué son los Signals en Angular?



Los **Signals** son una **nueva primitiva reactiva** introducida en Angular 16. Permiten **almacenar un valor y notificar automáticamente** cuando este cambia.

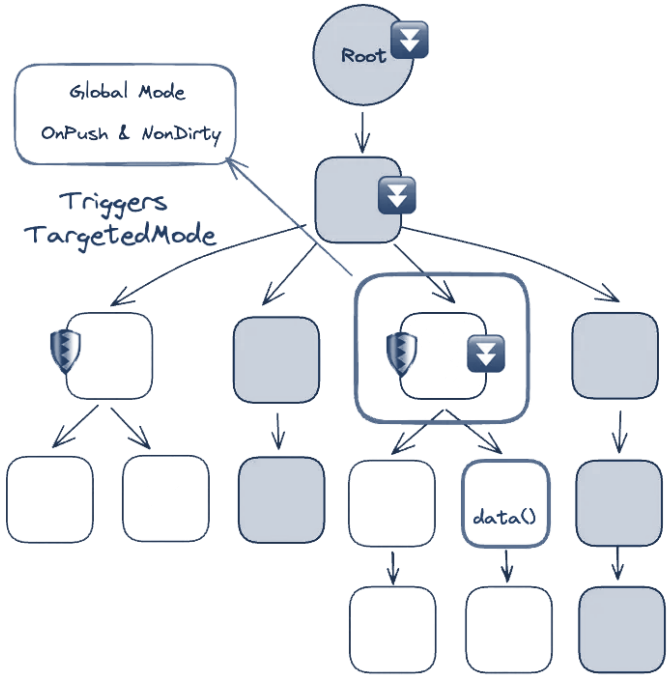
En Angular 21, los Signals ya están **totalmente integrados** en el framework y sus APIs modernas.

Su objetivo principal es ofrecer:

- **Reactividad más simple**: menos boilerplate que RxJS.
- **Mayor predictibilidad**: sin sorpresas con suscripciones manuales.
- **Eficiencia**: optimizan la detección de cambios eliminando el exceso de renders con Zone.js.

Base fundamental para la evolución de Angular: **SSR, SSG, Hydration y Router funcional** se apoyan en Signals.

Cómo Signals reemplaza a Zone.js



Problema con Zone.js

- Detecta **todos los cambios en la app**, incluso cuando no es necesario.
- Genera **muchos renders innecesarios,** impacto en rendimiento.
- Aumenta la **complejidad de depuración**.

Solución con Signals

- Cada signal **sabe quién depende de él**.
- Cuando cambia, **solo actualiza la parte de la UI vinculada**.
- Permite un **control granular y predecible** de la reactividad.
- Mejora el rendimiento al **eliminar renders innecesarios**.

- **Con Zone.js**:

Cambia una variable, toda la vista se vuelve a comprobar.

- **Con Signals**:

Cambia un signal, solo se repinta lo que depende de ese valor.

¿Cómo funciona Signal?

Las **variables normales** guardan un valor, pero **no reaccionan** cuando ese valor cambia.

Un **Signal = valor + notificación de cambio**.

Principales beneficios:

- **Mejor detección de cambios;** solo se actualiza lo necesario.
- **Código más simple;** sin suscripciones ni async pipe en exceso.
- **Mayor reactividad;** la UI se ajusta automáticamente al estado.

Metáfora: *"Un Signal es como una caja que guarda un valor y **brilla cada vez que cambia**, avisando a todo lo que depende de ella"*.

Creando y leyendo Signals

- **Crear un signal**: inicializa con un valor.

```
const counter = signal(0);
```

- **Leer un signal**: se accede como si fuera una función.

```
console.log(counter()); // 0
```

- **Actualizar un signal**:

```
// asigna un nuevo valor
counter.set(1);

// actualiza basado en el valor actual
counter.update((c: number) ⇒ c + 1);
```

- **Usar en la vista**: se interpola directamente.

```
<p>{{ counter() }}</p>
```

Signals derivados y enlazados

- **computed()**: crea un signal de solo lectura que **se recalcula automáticamente** cuando cambian sus dependencias.

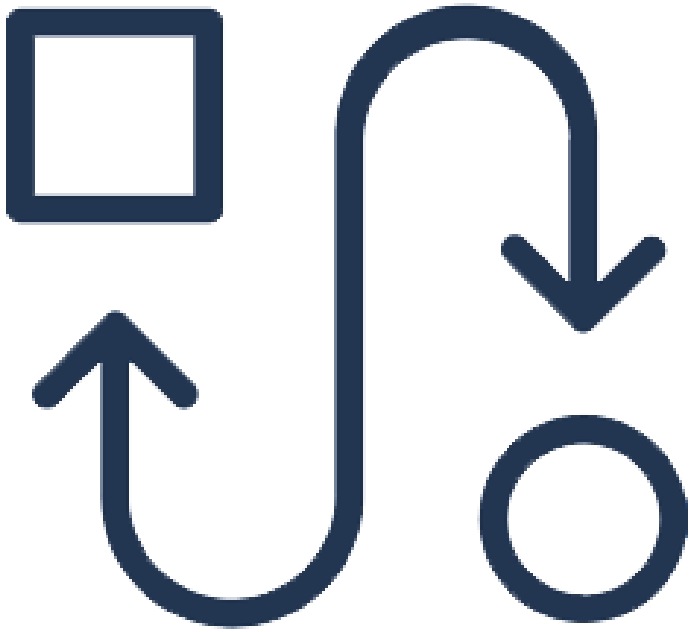
```
const price = signal(10);
const qty = signal(0);

// Computed: solo cálculo, sin modificar nada
const total = computed(() => {
  if (qty() === 0) return 0;      // condicional
  return price() * qty();
});

console.log(total()); // 0
qty.set(3);
console.log(total()); // 30
```

Signals derivados y enlazados

- **linkedSignal()**: crea un signal **escribible** que puede **resetearse o recalcularse** en base a otro signal.

```
const reset = signal(false);

// Linked: depende de otro signal, pero es escribible
const score = linkedSignal(() => {
  if (reset()) return 0;   // condicional de reseteo
  return 10;
});

console.log(score()); // 10
reset.set(true);
console.log(score()); // 0

// A diferencia de computed, aquí puedo modificarlo
score.set(50);
console.log(score()); // 50
```

Effect: reaccionando a los cambios



- **effect()** ejecuta una función **cuando cambian los signals que usa dentro**.
- Se usa para **efectos secundarios** (ej. logging, llamadas a servicios, actualización externa).
- Angular rastrea automáticamente las dependencias (los signals leídos dentro del effect).
- No devuelve valor, **su propósito es la acción**, no el cálculo.

```
const counter = signal(0);

// Effect: reacciona a cambios en counter
effect(() => {
  console.log(`El contador ahora es: ${counter()}`);
  localStorage.setItem('lastCounter', counter().toString());
});

// Cambios en el signal
// Consola: "El contador ahora es: 1"
counter.set(1);
// Consola: "El contador ahora es: 6"
counter.update((c: number) => c + 5);
```

Recuperando datos con Signals

Antes se usaban **Observables** con suscripción manual y unsubscribe. Con Angular 20 aparece la **Resource API (httpResource)** como alternativa.

Permite:

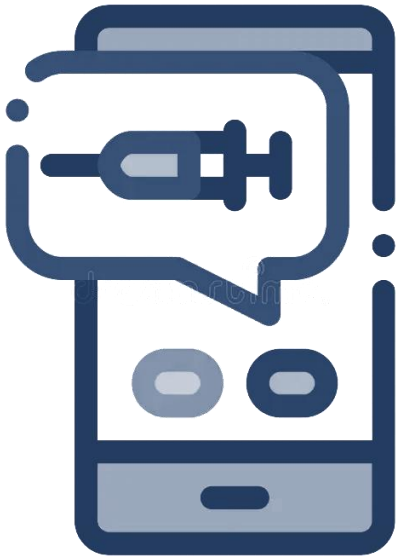
- Cargar datos directamente en un **signal**.
- Consultar el **estado de la petición**: valor, carga, error.
- Recargar los datos fácilmente con reload().

Aporta un modelo más **declarativo y reactivo** para trabajar con datos remotos.

NOTA: AÚN SE CONSIDERA EXPERIMENTAL, POR LO QUE SU API PUEDE CAMBIAR EN VERSIONES FUTURAS.

05 - INYECCIÓN DE DEPENDENCIAS MODERNA EN ANGULAR

¿Qué es la Inyección de Dependencias (DI)?



DI es el mecanismo que permite:

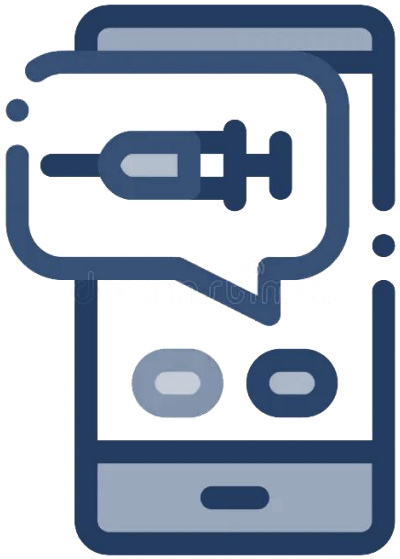
- Separar responsabilidades
- Desacoplar clases
- Reutilizar servicios
- Facilitar testing

Angular tiene un **Injector jerárquico**.

No creamos dependencias manualmente.

Angular las provee.

Antes vs Ahora



Antes (Angular clásico):

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

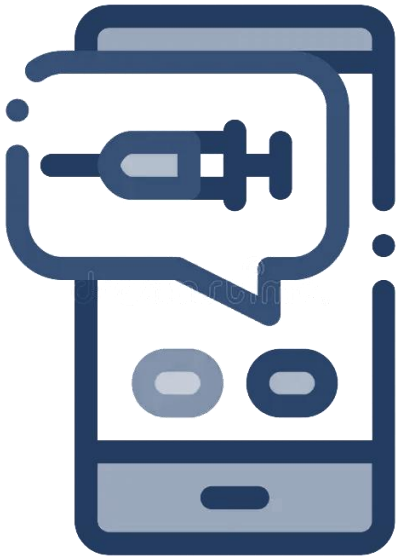
Ahora (Angular 21 moderno):

```
private http = inject(HttpClient);
```

Ventajas:

- Menos boilerplate
- No necesitas constructor
- Más limpio en standalone

Providers en Angular



En standalone, los providers se configuran en:

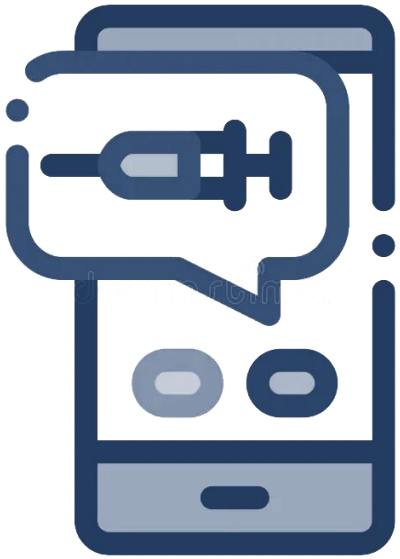
```
bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient()
  ]
});
```

Ya no necesitas AppModule.
También puedes registrar providers por componente:

```
@Component({
  standalone: true,
  providers: [MyService]
})
```

Angular crea un scope local.

Alcance del Injector



Angular tiene jerarquía:

- Root injector (global)
- Injector de componente
- Injector de directiva

Si registras un provider en root:

```
@Injectable({ providedIn: 'root' })
```

Es singleton global.

Si lo registras en un componente:
Se crea instancia por componente.

Esto es clave para diseño correcto.

06 – ESTRUCTURA BASE DEL PROYECTO

Arquitectura Frontend Angular

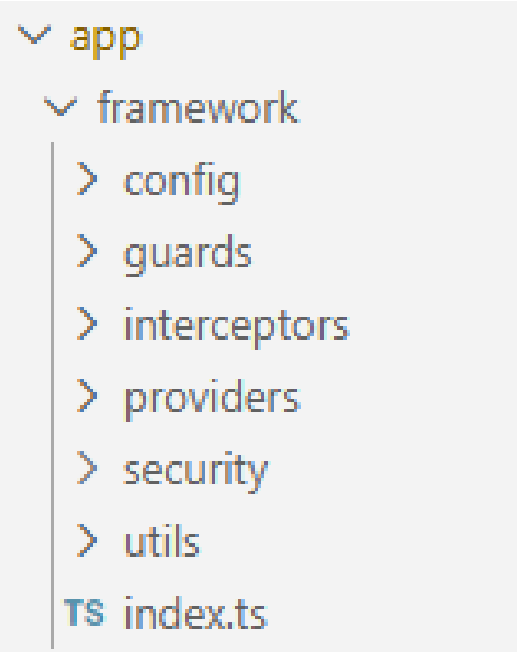
La aplicacion frontend sigue una arquitectura modular basada en Clean Architecture y Domain-Driven Design (DDD) adaptados al ecosistema Angular. El sistema se compone de tres niveles:

1. **Aplicacion principal** (ng-library-client): aplicacion Angular con SSR que actua como host, gestiona el routing global, la seguridad, el layout y la composicion de modulos de negocio.
2. **Librerias de dominio** (dentro de projects/): cada modulo de negocio es una Angular library independiente con capas DDD completas (Domain, Application, Infrastructure, Presentation). Se publican como entry points del workspace principal.
3. **Librerias de arquitectura** (eac-arch-*): paquetes NPM independientes que proporcionan los building blocks, abstracciones e infraestructura transversal. Cada una tiene su propio repositorio Git y ciclo de vida de publicacion.

Esta separacion permite que los equipos trabajen de forma autonoma en cada modulo, reutilicen las abstracciones arquitectonicas y mantengan una estructura consistente en todos los bounded contexts.

Estructura Base del Proyecto

Aplicacion Principal - Framework Layer



La capa framework/ contiene toda la infraestructura transversal de la aplicación. Es el punto de arranque y configuración.

config/: configuración de la aplicación Angular. Registra todos los providers globales: zoneless change detection, router, client hydration con event replay para SSR, configuración externa (provideConfig()), seguridad OAuth (provideSecurity()), y HTTP client con interceptores de autenticación (authInterceptor).

guards/: protección de rutas. Se usa authGuard de la librería de seguridad para proteger las rutas que requieren autenticación. Las rutas bajo /library estan protegidas tanto a nivel padre como hijos.

interceptors/: interceptores HTTP. El authInterceptor inyecta automáticamente el token de acceso en cada petición HTTP saliente.

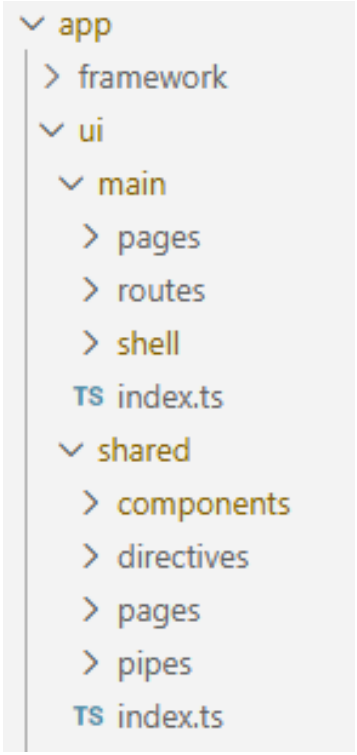
providers/: proveedores de inyección de dependencias global.

security/: utilidades de seguridad a nivel de la aplicación host.

Características técnicas: Angular Zoneless (sin Zone.js), SSR con hydration y event replay, y configuración externa inyectada en runtime.

Estructura Base del Proyecto

Aplicacion Principal - UI Layer



La capa ui/ organiza toda la presentación visual en dos bloques: main/ (especifico de la app) y shared/ (reutilizable).

main/shell/: contiene el componente raíz App (solo un RouterOutlet), el Layout (estructura con sidenav responsivo usando Angular Material CDK BreakpointObserver, signals y computed properties), el Header y el Nav.

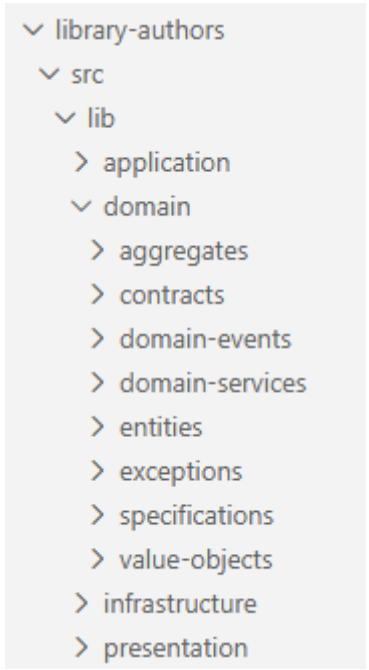
main/routes/: configuración de rutas globales. La ruta raíz carga Home, las rutas /library/* cargan el Layout protegido con authGuard y usan lazy loading para las librerías de dominio (loadChildren: () => import('library-authors')). Incluye ruta de callback OAuth (/security/auth/callback) y fallback 404.

main/pages/: paginas principales como Home.

shared/components/: componentes reutilizables: ThemeToggle (servicio de temas con signals: default/orange/purple + modo oscuro), ConfigViewer (dialog para inspeccionar configuración en runtime), TokenInfo (dialog para inspeccionar tokens OAuth).

shared/pages/: pagina PageNotFound para rutas no encontradas.
shared/directives/: directivas compartidas.
shared/pipes/: pipes compartidos.

Librerias de Dominio - Capa Domain



Cada librería de dominio (ejemplo: library-authors) implementa DDD táctico completo. La capa Domain es el núcleo, sin dependencias externas excepto el Shared Kernel.

aggregates/: Aggregate Roots. Author extiende BaseAggregateRoot del shared kernel. Constructor privado, factory method create(), y mutaciones explícitas (updateProfile(), updateName(), updateLifeSpan(), changeGenre()). Encapsula la identidad y compone Value Objects. Todas las modificaciones pasan por métodos que fuerzan invariantes.

value-objects/: objetos inmutables sin identidad. FullName (nombre y apellido, igualdad case-insensitive), LifeSpan (fecha nacimiento/muerte, cálculo de edad, estatus de fallecimiento), LiteraryGenreRef (referencia a catálogo, igualdad por id). Todos extienden ValueObject del shared kernel, implementan getEqualityComponents() y se congelan con Object.freeze().

exceptions/: excepciones de dominio con validación integrada. InvalidFullNameException, InvalidLifeSpanException, InvalidLiteraryGenreException. Cada una extiende ValidationException y tiene un método validate() estático que recolecta errores por campo antes de lanzar.

Estructura Base del Proyecto

Librerías de Dominio - Capa Application

La capa Application orquesta los casos de uso y define los contratos (puertos) siguiendo el patrón CQRS.

use-cases/: separados en commands/ (operaciones de escritura) y queries/ (operaciones de lectura). Cada use case implementa CommandUseCase o QueryUseCase del shared kernel, recibiendo un command/query tipado y retornando un Observable.

contracts/: interfaces que definen los puertos de la capa de aplicación. Separadas en persistence/ (contratos de repositorio y persistencia) y queries/ (contratos de servicios de consulta).

models/: DTOs y View Models que representan los datos de aplicación.

mappings/: transformaciones entre DTOs de infraestructura, modelos de aplicación y entidades de dominio.

exceptions/: excepciones específicas de la capa de aplicación.

Librerías de Dominio - Capa Infrastructure

La capa Infrastructure implementa los adaptadores concretos, conectando el dominio con el mundo exterior.

rest-clients/: clientes HTTP tipados. AuthorsHttpClient es un servicio Angular inyectable que usa HttpClient para comunicarse con la API REST. Construye HttpParams dinámicos, parsea el header X-Pagination para metadatos de paginación, y deserializa las respuestas en DTOs tipados. Usa ConfigService para obtener la URL base en runtime.

rest-clients/contracts/: interfaces que tipan las peticiones y respuestas HTTP (GetAllAuthorsHttpRequest/Response, GetAuthorByIdHttpRequest/Response).

rest-clients/dtos/: DTOs que reflejan exactamente la API del servidor. AuthorDto replica los campos del backend. PaginationMeta replica la estructura del header X-Pagination.

persistence/mappings/: mappers entre DTOs de infraestructura y entidades/modelos de dominio.

agents/: adaptadores adicionales (contratos y DTOs para integraciones con agentes externos).

Estructura Base del Proyecto

Librerías de Dominio - Capa Presentation

La capa Presentation es la interfaz de usuario del módulo, completamente autocontenida.

shell/: componente contenedor del módulo (AuthorsShell). Usa Angular Material (cards, icons, buttons, menus) y RouterOutlet para renderizar las páginas hijas. Actúa como layout propio del módulo dentro del layout global de la app.

routes/: configuración de rutas internas del módulo (AUTHORS_ROUTES). Define el shell como componente padre y las páginas como hijos. Estas rutas se cargan con lazy loading desde el router de la aplicación principal.

pages/: páginas del módulo (por ejemplo AuthorListPage). Usa ChangeDetectionStrategy.OnPush para optimizar el rendimiento.

components/: componentes visuales específicos del módulo.
El módulo se integra con la app principal vía lazy loading. Esto permite carga bajo demanda y aislamiento completo del bundle.

Shared Kernel - Building Blocks DDD

La librería **eac-arch-shared-kernel** proporciona las abstracciones base que garantizan consistencia en todos los bounded contexts.

Domain Layer:
- Entity: contrato base con identidad. Funciones auxiliares isTransient() y entitiesEqual() para comparación por identidad.
- ValueObject: clase abstracta con igualdad estructural vía getEqualityComponents(). Dos value objects son iguales si todos sus componentes coinciden.
- AggregateRoot: contrato y clase base BaseAggregateRoot con gestión de domain events (addDomainEvent(), clearDomainEvents()).
- DomainEvent: contrato con eventId, occurredOn, eventVersion, correlationId. Factory createDomainEvent() con UUID y timestamp automáticos.
- Specification: patrón Specification composable con CompositeSpecification que soporta and(), or() y not().

Application Layer:
- Command / CommandUseCase / CommandHandler: lado de escritura CQRS.
- Query / QueryUseCase: lado de lectura CQRS.
- Request: contrato base para commands y queries.
- Result: wrapper de respuesta con success, data, errors. Factories successResult() y failResult().
- ReadModel: marcador para DTOs del lado de lectura.
- PagedList: colecciones paginadas con currentPage, totalPages, pageSize, totalCount, hasPrevious, hasNext. Factories createPagedList() y emptyPagedList().
- Persistence: contratos ReadOnlyRepository, GenericRepository, UnitOfWork, QueryService.
- DataShaping: sparse fieldsets con ShapeData, shapeEntity(), shapeCollection().

Exceptions: jerarquía de excepciones tipadas: BaseException (con ExceptionError[] estructurados por campo), NotFoundException, ConflictException, ValidationException, UnprocessableEntityException, InvalidSortParameterException.

Utils: calculateAge(), SortField con parseSortFields() y applySorting(), PatchOperation (JSON Patch RFC 6902) con UpdatePartialDelta.

Librerías de Infraestructura Transversal

Paquetes NPM independientes que implementan las preocupaciones transversales:

- **@eac-arch/infrastructure-config**: gestión de configuración externa. Provee ConfigService para obtener valores en runtime. Se registra con provideConfig().
- **@eac-arch/infrastructure-security**: autenticación y autorización OAuth/OIDC (Keycloak). Provee authGuard para protección de rutas, authInterceptor para inyección automática de tokens, OidcCallback como componente de callback OAuth, y provideSecurity() para registrar el sistema completo.
- **@eac-arch/infrastructure-http**: clientes HTTP base e interceptores reutilizables.
- **@eac-arch/infrastructure-persistence**: persistencia en el lado del cliente (LocalStorage, IndexedDB). Implementa los contratos de repositorio del shared kernel.
- **@eac-arch/infrastructure-state**: gestión de estado reactiva (store patterns). Centraliza el estado de la aplicación.
- **@eac-arch/infrastructure-telemetry**: telemetría y observabilidad. Captura métricas, logs y trazas del frontend.
- **@eac-arch/ui-kit**: componentes visuales reutilizables basados en Angular Material. Librería de diseño común para todos los módulos.

07 - ANGULAR MATERIAL + THEMING BASE

¿Qué es Material Design?

Material Design no es una librería.

Es un **Design System completo** creado por Google en 2014 y evolucionado a **Material 3**.

Un Design System define:

- Lenguaje visual
- Sistema de componentes
- Sistema de interacción
- Sistema de tokens (colores, tipografía, espaciado)
- Reglas de comportamiento

Es arquitectura visual, no decoración.

Problema que Resuelve

Sin sistema de diseño:

- Cada equipo diseña distinto
- UI inconsistente
- Mala experiencia
- Alto costo de mantenimiento

Con Material:

- Coherencia transversal
- Decisiones centralizadas
- Escalabilidad visual
- Accesibilidad integrada

En empresas grandes esto es crítico.

Material 3

Material 3 introduce:

- Dynamic color system
- Design tokens
- Mejor accesibilidad
- Adaptabilidad a dispositivos
- Mayor personalización

Ya no es rígido como Material 2.
Es más flexible y empresarial.

Elementos Fundamentales de Material

Material define:

1. **Color System**
   - Primary
   - Secondary
   - Tertiary
   - Error
   - Surface
2. **Typography Scale**
   - Display
   - Headline
   - Title
   - Body
   - Label
3. **Elevation**
   - Sombras como jerarquía visual.
4. **Motion**
   - Animaciones coherentes.

Esto no lo decide cada desarrollador.
Está estandarizado.

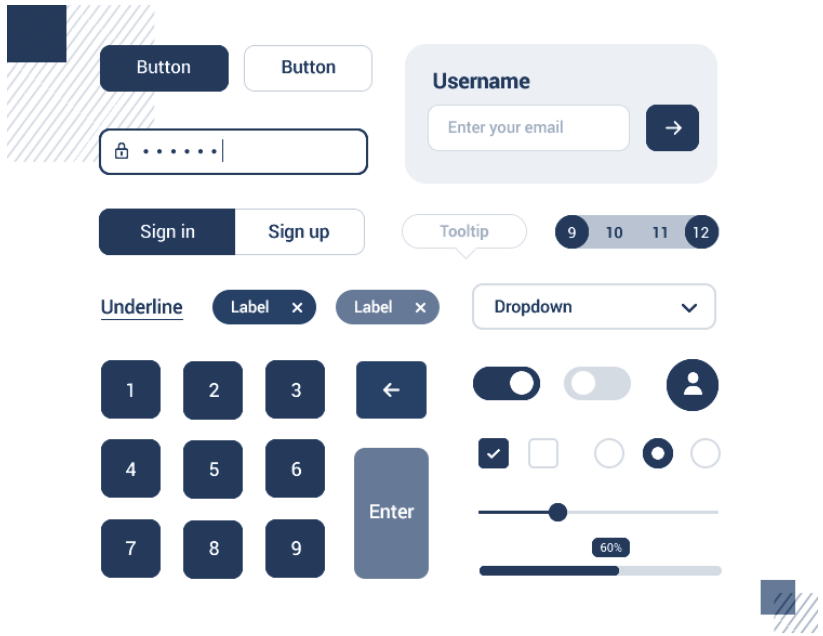
Material como Arquitectura Visual

Material actúa como:

Arquitectura técnica: Clean / DDD
Arquitectura visual: Material

Ambas deben ser coherentes.
Una mala UI puede arruinar una buena arquitectura.

¿Qué es Angular Material realmente?

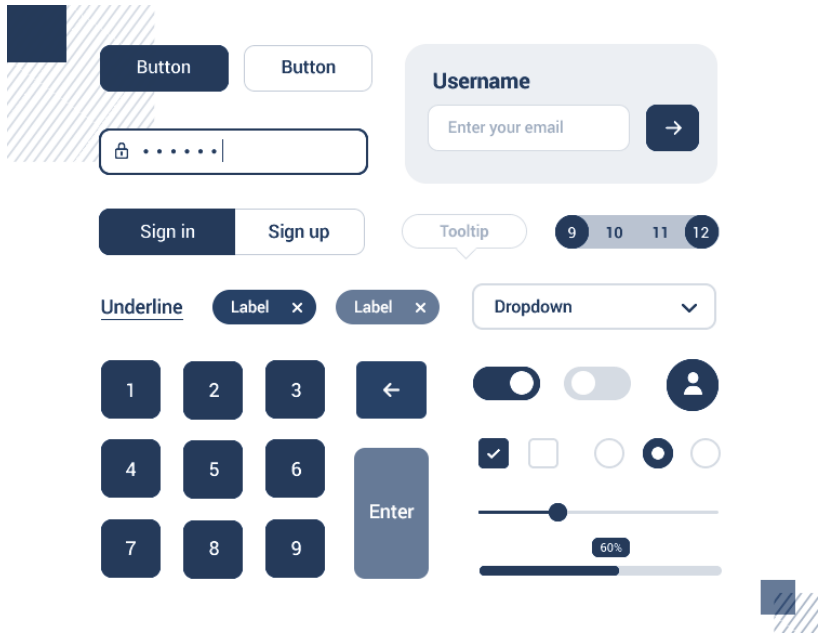


Angular Material es:

- Una librería oficial mantenida por el equipo Angular.
- Basada en Material Design 3.
- Construida sobre Angular CDK.
- Totalmente integrada con el ciclo de vida y el change detection de Angular.
- Pensada para accesibilidad y consistencia empresarial.

No es Bootstrap.
No es solo CSS.
Es infraestructura UI.

Arquitectura Interna de Angular Material



Angular Material se construye en dos capas:

1. **Angular CDK (Component Dev Kit)**

Infraestructura sin estilos:

- Overlay system
- Portals
- Scrolling
- A11y (accesibilidad)
- Drag & Drop
- Virtual scrolling
- Focus management
- CDK es la base técnica.

2. **Componentes Material**

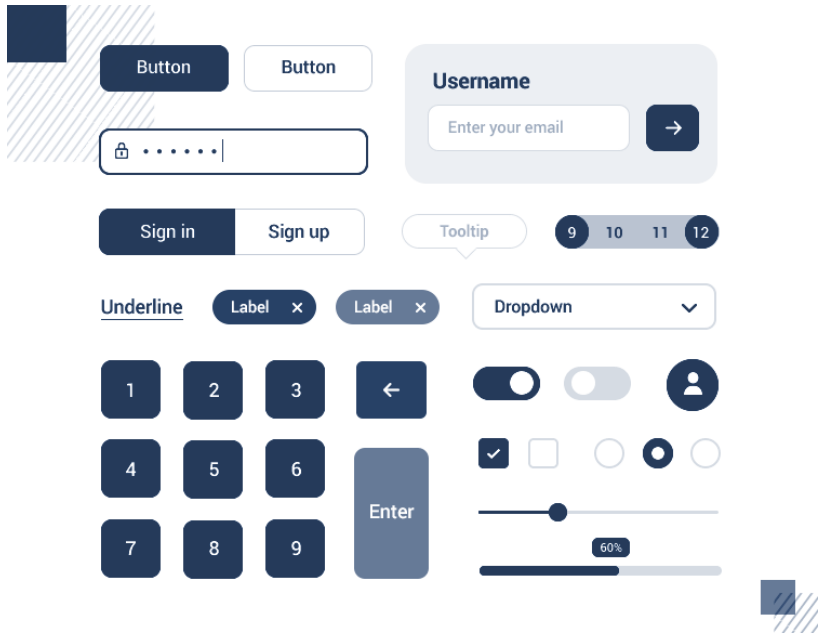
Encima del CDK se construyen:

- MatButton
- MatDialog
- MatTable
- MatToolbar
- MatSidenav
- MatFormField

etc.

Los componentes usan CDK internamente.

¿Por qué Angular Material es empresarial?

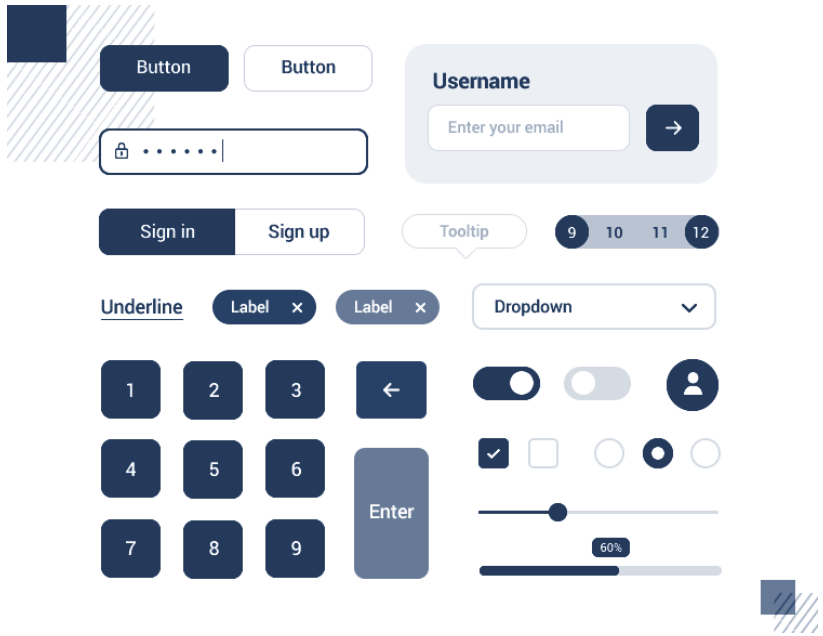


Porque garantiza:

- Accesibilidad (ARIA roles automáticos)
- Teclado y focus management correcto
- Performance optimizada
- Estructura consistente
- Animaciones coherentes

En un entorno corporativo esto no es opcional.

Integración con Angular



En Angular moderno:
No necesitas NgModule global.
Puedes importar directamente:

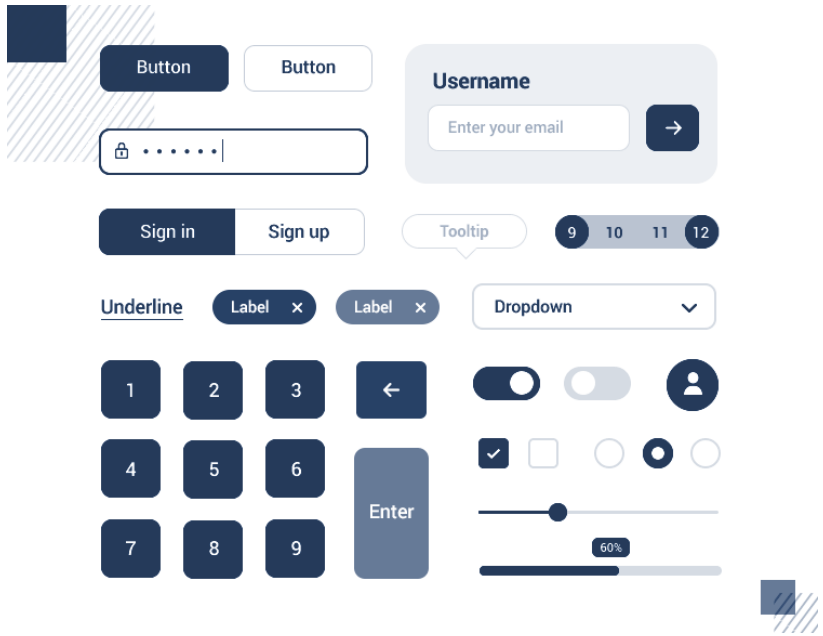
```
@Component({
    standalone: true,
    imports: [MatButtonModule, MatCardModule]
})
```

Esto hace:

- Menor acoplamiento
- Mejor tree shaking
- Mejor modularidad

Angular Material está completamente adaptado a standalone.

Theming Técnico en Angular Material 3



Angular Material 3 usa:

- Design tokens
- CSS variables
- Paletas dinámicas

El theme se define en SCSS:

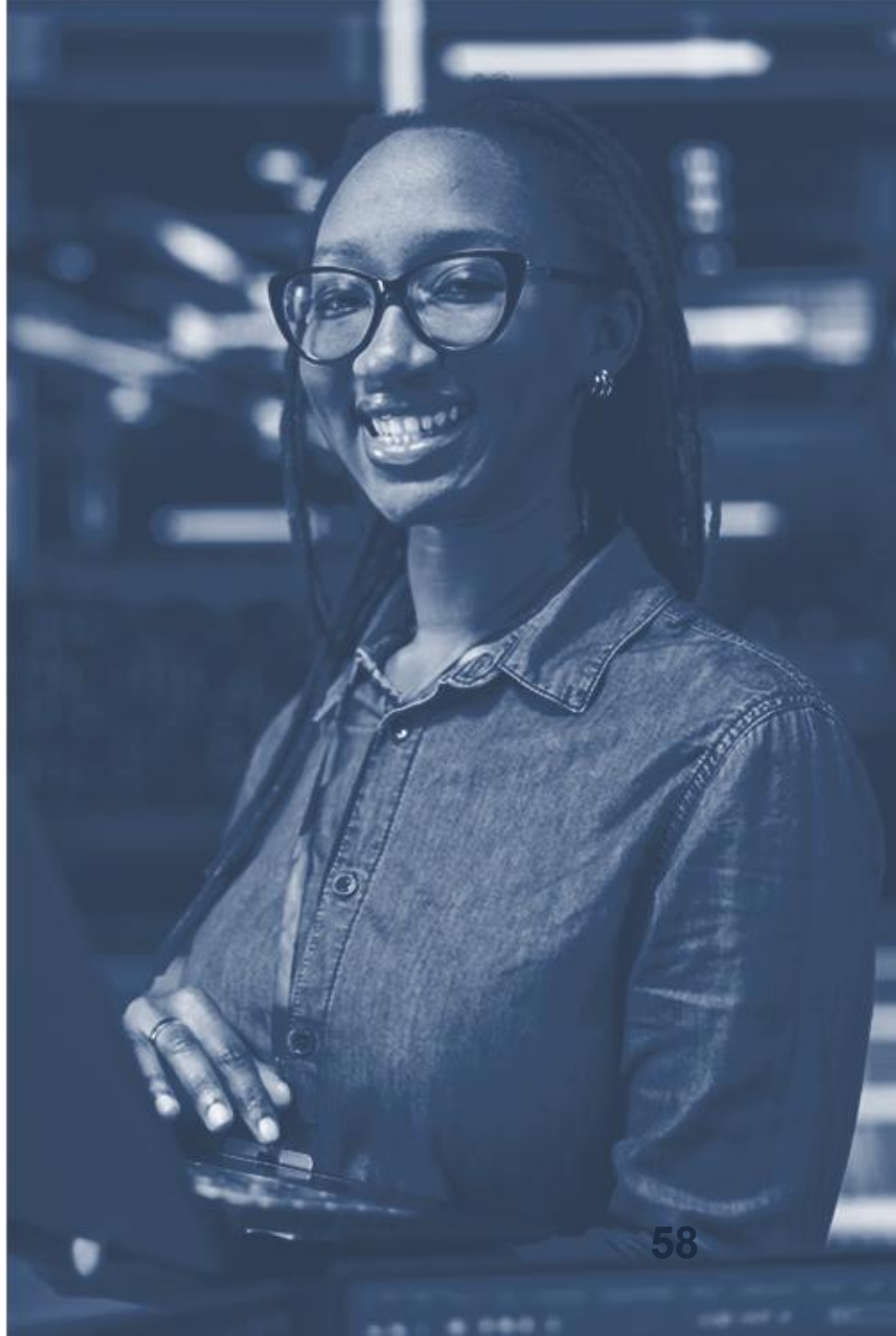
```
@use '@angular/material' as mat;

$theme: mat.define-theme((
  color: (
    primary: mat.$azure-palette,
  )
));
```

Luego se aplica globalmente.

Beneficio: Cambias branding sin tocar cada componente.

GRACIAS

POR SU PREFERENCIA



GALAXY
TRAINING

Ing. ERICK ARÓSTEGUI CUNZA