GALAXY
TRAINING

Ing. ERICK ARÓSTEGUI CUNZA

Sesión 01

Arquitectura del Cliente en Sistemas SPA Modernos

Instructor

ERICK AROSTEGUI CUNZA

erick.arostegui.cunza@gmail.com

GALAXY TRAINING

ARQUITECTURA DE APLICACIONES

MULTI-STACK AVANZADO

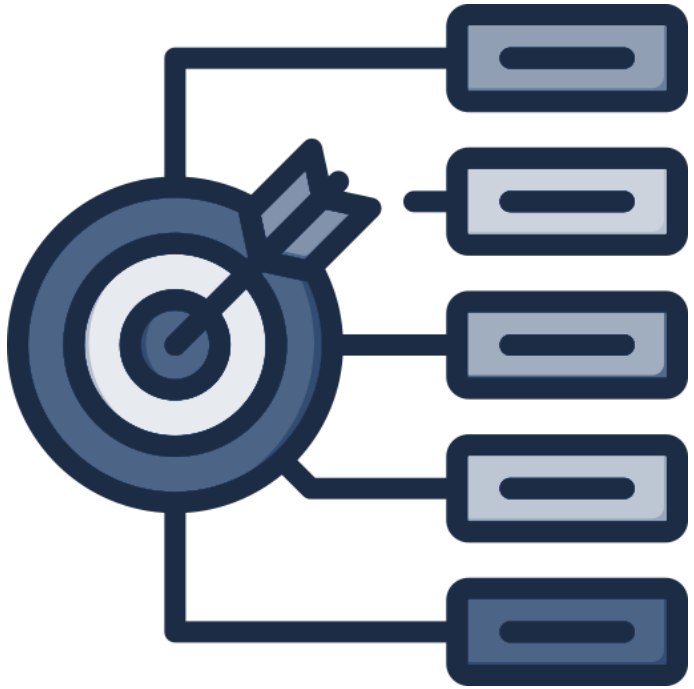
Angular 21

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

01 - CONTEXTO Y OBJETIVO DEL MÓDULO

Contexto y Objetivo del Módulo

¿Qué es exactamente el sistema Library?



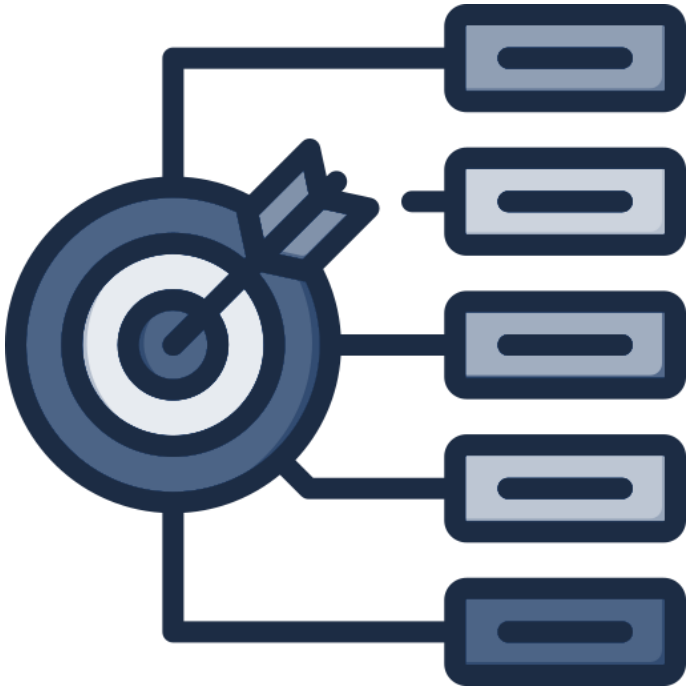
Library es un sistema de venta de libros que expone una API REST organizada en tres módulos de negocio:

Authors: Gestión de autores.
Books: Gestión de libros.
Catalog: Exposición y consulta estructurada del catálogo.

Cada módulo representa un dominio funcional distinto y está desacoplado en backend.

El cliente no está consumiendo endpoints aislados. Está interactuando con dominios bien definidos.

¿El Cliente Es Solo una Capa de Presentación?



No.

Aunque el backend concentra la lógica crítica, el frontend:

- Modela datos del dominio.
- Coordina casos de uso.
- Gestiona estado.
- Maneja validaciones.
- Controla navegación entre contextos.

Por lo tanto, el cliente también contiene complejidad estructural.

Si el backend tiene arquitectura y el frontend no, se rompe la coherencia del sistema completo.

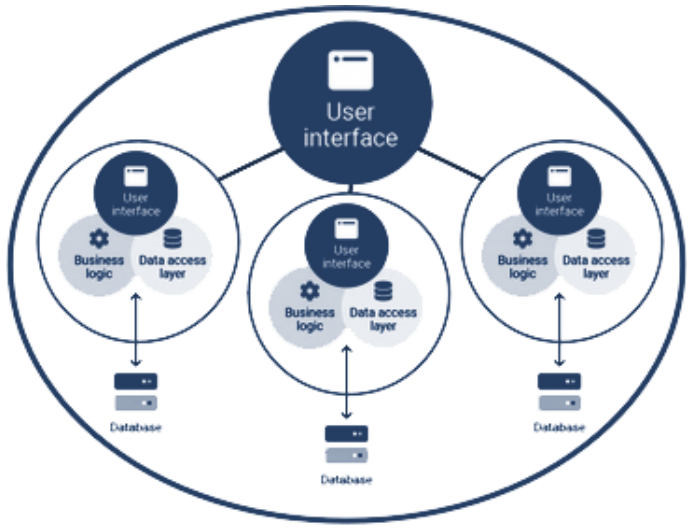
¿Qué pasa si el Cliente no respeta los Dominios?

Si el cliente no replica la separación Authors / Books / Catalog:

- Se mezclan responsabilidades.
- Se generan dependencias cruzadas.
- Cambios en Books impactan Authors.
- Catalog termina acoplado a lógica interna de otros módulos.
- El crecimiento se vuelve caótico.

El frontend debe respetar los límites del dominio.

¿Por qué elegimos Monolito Modular?



Modular Monolithic Architecture

¿Por qué no Micro-Frontends desde el inicio?

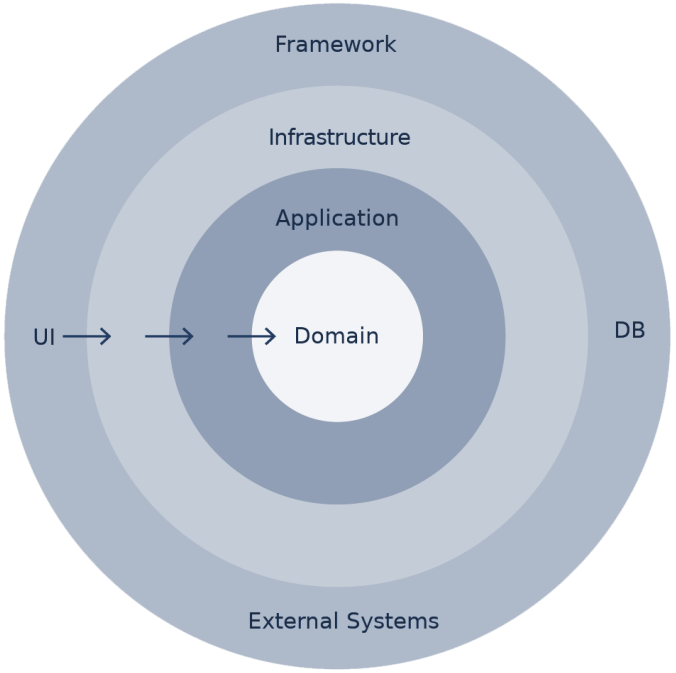
- Tenemos una sola aplicación.
- Los módulos están relacionados.
- No existe necesidad organizacional de despliegue independiente.
- Micro-Frontends añaden complejidad operativa.

Elegimos Monolito Modular porque:

- Permite separar dominios.
- Controla dependencias.
- Reduce complejidad inicial.
- Mantiene coherencia arquitectónica.
- Permite evolucionar a Micro-Frontends si el negocio lo requiere.

Decisión basada en contexto, no en tendencia.

¿Por qué aplicar Clean Architecture en el Cliente?

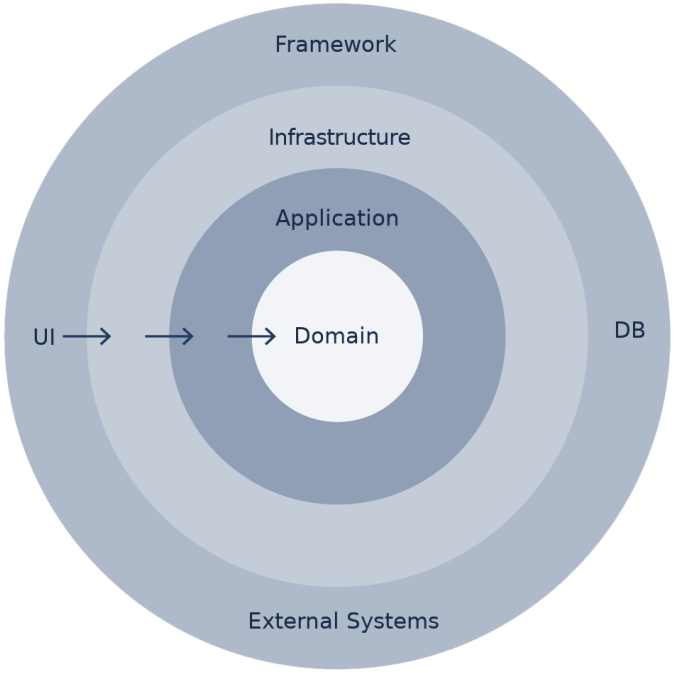


Si ya tenemos módulos (Authors/Books/Catalog), ¿no es suficiente?

Respuesta:
No necesariamente. En Angular puedes tener módulos y aun así terminar con esto:

- Componentes haciendo llamadas HTTP directas.
- Services “gigantes” mezclando: HTTP + mapeo + estado + lógica.
- DTOs del backend usados tal cual en la UI (acoplamiento al contrato REST).
- Cambios en la API rompiendo pantallas por todas partes.

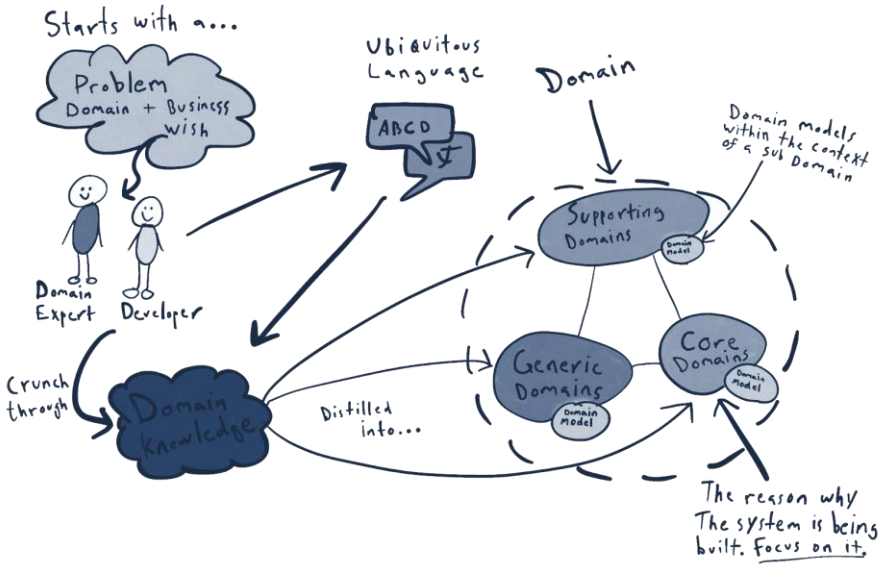
¿Por qué aplicar Clean Architecture en el Cliente?



¿Qué garantiza Clean *de forma realista* en Angular?

1. **La UI no conversa con HTTP**
   - Los componentes hablan con **casos de uso / facades** (Application).
   - HTTP queda encapsulado en **adaptadores** (Infrastructure).
2. **El módulo no se casa con el contrato REST**
   - La UI usa **models del cliente** (del módulo).
   - Los DTOs REST se mapean en un solo lugar (adapter/mapper).
3. **Lógica y reglas quedan fuera de componentes**
   - Validaciones, reglas, combinaciones de datos Application/Domain (según el nivel).
   - Component = render + eventos de usuario.
4. **Testeo útil sin depender del backend**
   - Puedes testear casos de uso con mocks del adapter HTTP.
   - No necesitas levantar la API para validar comportamiento básico.

¿Por qué usar DDD en el Frontend?



¿DDD no es solo para backend?
No.

DDD no es una tecnología, es una forma de organizar el sistema alrededor del **dominio del negocio**.

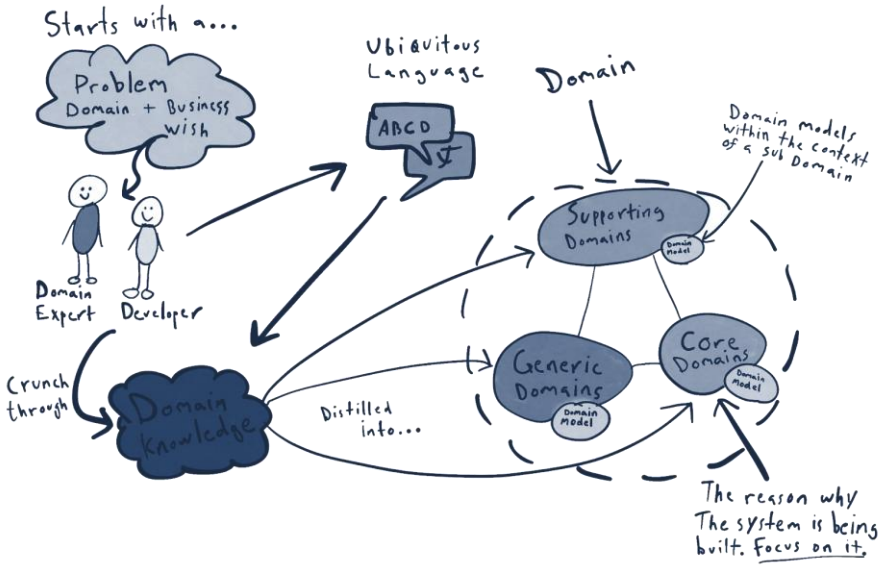
Si el backend de Library ya está dividido en:

- Authors
- Books
- Catalog

Entonces el frontend que consume esa API **también está interactuando con esos dominios**, no con “pantallas sueltas”.

Contexto y Objetivo del Módulo

¿Por qué usar DDD en el Frontend?



1. **Respetar los dominios existentes**
   - Authors, Books y Catalog no se mezclan.
   - Cada uno mantiene sus propios modelos y casos de uso.
2. **Mantener límites claros**
   - Un módulo no accede directamente a la lógica interna de otro.
   - La comunicación ocurre a través de contratos definidos.
3. **Usar el mismo lenguaje que el backend**
   - El frontend habla de Author, Book y Catalog.
   - No de “data genérica” o servicios ambiguos.
4. **Evitar servicios transversales gigantes**
   - No existe un “LibraryService” que haga todo.
   - Cada contexto es responsable de su propio comportamiento.

Contexto y Objetivo del Módulo

¿Cuál es el objetivo arquitectónico del Módulo Cliente?

Al finalizar el módulo, el cliente debe:

- Estar organizado por dominios.
- Mantener dependencias controladas.
- Separar presentación de lógica.
- Estar preparado para crecer.
- Poder evolucionar a distribución futura si el negocio lo exige.

No buscamos solo funcionalidad.
Buscamos arquitectura sostenible.

02 – TIPOS DE APLICACIONES WEB

¿Qué tipo de aplicación es Library?

Desktop

Tablet

Mobile

Smart Watch

Antes de hablar de arquitectura, debemos responder:

¿Estamos construyendo una landing, un sitio estático o un sistema interactivo?

Library es:

- Un sistema de venta de libros.
- Con múltiples módulos funcionales.
- Con operaciones CRUD.
- Con navegación entre contextos.
- Con interacción continua con una API REST.

Conclusión:
No es un sitio informativo.
Es una aplicación interactiva de negocio.

Tipos de aplicaciones Web (Panorama General)

Existen varios tipos de aplicaciones web:

- Sitios informativos / Landing
- SPA (Single Page Application)
- SSR (Server-Side Rendering)
- SSG (Static Site Generation)
- PWA (Progressive Web App)
- Micro-Frontends (modelo organizacional)

Cada tipo responde a necesidades distintas.
La arquitectura depende del contexto.

Sitio Informativo / Landing

¿Qué es?
- Sitio principalmente estático.
- Contenido público.
- Poca o nula interacción con backend.
- Objetivo principal: visibilidad y marketing.

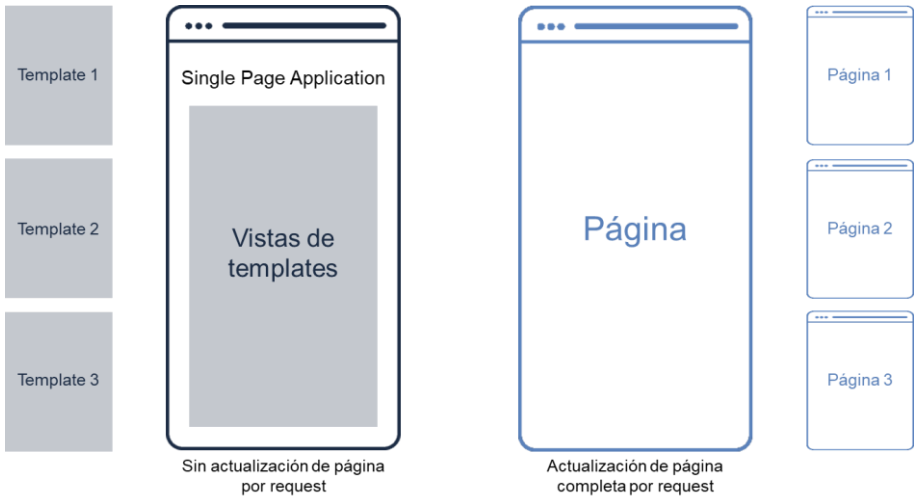
¿Cuándo se usa?
- Web corporativa.
- Página de producto.
- Blog informativo.

¿Por qué Library no es esto?
- Library no es contenido estático.
- Requiere autenticación.
- Tiene operaciones CRUD.
- Interactúa constantemente con una API REST.

No estamos construyendo una landing.

Tipos de Aplicaciones Web

SPA (Single Page Application)



¿Qué es?

- Aplicación que carga una vez.
- Navegación sin recarga completa.
- Estado gestionado en cliente.
- Comunicación constante con API.

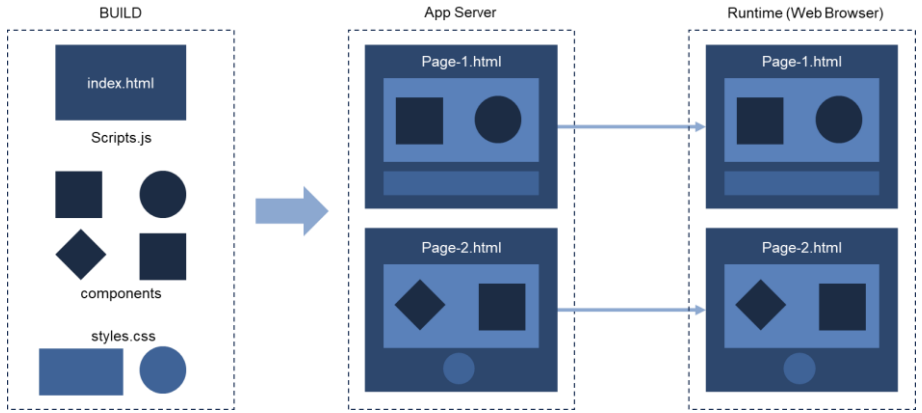
¿Cuándo se usa?

- Sistemas internos.
- Paneles administrativos.
- Aplicaciones empresariales.

¿Por qué Library encaja aquí?

- Navegación entre Authors, Books y Catalog.
- Operaciones dinámicas.
- Interacción frecuente con backend.
- Experiencia fluida requerida.

SSR (Server-Side Rendering)



¿Qué es?

- Renderizado inicial en servidor.
- Mejora SEO.
- Mejor tiempo de primer render para contenido público.

¿Cuándo se usa?

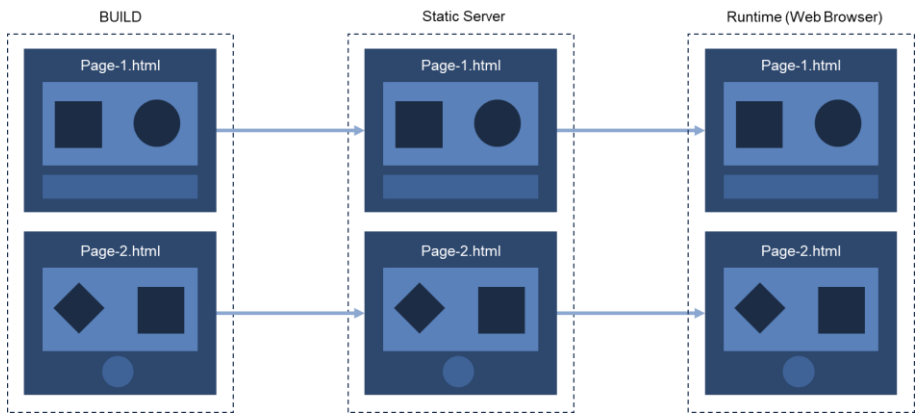
- E-commerce público.
- Portales con fuerte dependencia de buscadores.
- Contenido indexable.

¿Por qué no es prioritario en Library?

- Library es principalmente autenticado.
- No depende críticamente del SEO.
- Es más importante la interacción que la indexación.

SSR no responde al problema principal de Library.

SSG (Static Site Generation)



¿Qué es?

- Generación de HTML estático en build.
- Alto rendimiento.
- Ideal para contenido que cambia poco.

¿Cuándo se usa?

- Documentación.
- Blogs.
- Sitios institucionales.

¿Por qué no aplica a Library?

- Library tiene datos dinámicos.
- CRUD constante.
- Contenido generado por usuarios.

SSG no encaja con un sistema transaccional.

PWA (Progressive Web App)

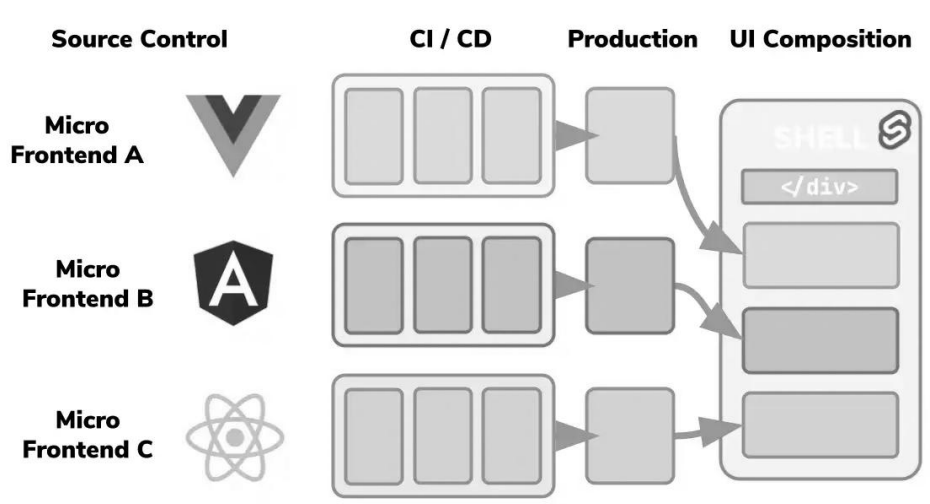
¿Qué es?
- Aplicación web con capacidades offline.
- Instalación tipo app.
- Uso de service workers.

¿Cuándo se usa?
- Aplicaciones móviles ligeras.
- Sistemas con conectividad intermitente.
- Experiencias tipo app sin tienda.

¿Aplica a Library?
- Podría aplicarse en el futuro.
- No es requisito actual.
- No resuelve el problema arquitectónico central.

No es prioridad en esta etapa.

Micro-Frontends



¿Qué es?
- Aplicación dividida en múltiples frontends independientes.
- Despliegue separado por dominio.
- Alta autonomía organizacional.

¿Cuándo se usa?
- Equipos grandes.
- Múltiples releases independientes.
- Alta escala organizacional.

¿Por qué no ahora?
- Library es una sola aplicación.
- No existe necesidad de despliegue independiente.
- Añadiría complejidad innecesaria.

Primero disciplina modular, luego distribución si el negocio lo exige.

03 - CRITERIOS PARA ELEGIR EL TIPO DE APLICACIÓN

Criterios para Elegir el Tipo de Aplicación

¿Cómo se decide el tipo de Aplicación?

La elección del tipo de aplicación no es técnica.
Es contextual.

Se basa en:

- Naturaleza del negocio.
- Tipo de usuario.
- Nivel de interacción.
- Necesidades organizacionales.
- Proyección de crecimiento.

En Library analizamos esos factores antes de elegir.

Criterios para Elegir el Tipo de Aplicación

Criterio 1: Naturaleza del Sistema

¿Es contenido público o sistema interactivo?

Library:

- Requiere autenticación.
- Permite crear y editar entidades.
- Navega entre módulos.
- Ejecuta operaciones frecuentes contra API.

No es un sitio informativo.
Es un sistema interactivo transaccional.

Eso descarta SSG y Landing.

Criterios para Elegir el Tipo de Aplicación

Criterio 2: SEO vs Interacción

Cuando el SEO es crítico:

- SSR o SSG tienen sentido.

Cuando la interacción es crítica:

- SPA es más adecuada.

Library:

- No depende del posicionamiento público.
- Depende de experiencia fluida.
- Prioriza productividad y navegación rápida.

La prioridad es interacción, no indexación.

Criterios para Elegir el Tipo de Aplicación

Criterio 3: Complejidad Organizacional

¿Necesitamos despliegue independiente por módulo?

En Library:

- Es una sola aplicación.
- Los módulos comparten sesión y contexto.
- No existen equipos completamente aislados.

Micro-Frontends introduciría complejidad innecesaria.

Primero modularidad interna, luego distribución si el negocio lo requiere.

Criterios para Elegir el Tipo de Aplicación

Criterio 4: Escalabilidad Futura

Escalabilidad no significa solo volumen de usuarios.

Significa:

- Nuevos módulos.
- Nuevas funcionalidades.
- Nuevos equipos.
- Evolución tecnológica.

La arquitectura elegida debe permitir crecer sin reestructurar todo.

Por eso:

- SPA
- Modular
- Clean
- DDD

Criterios para Elegir el Tipo de Aplicación

Decisión Fundamentada para Library

Después de evaluar:

- Tipo de sistema
- Nivel de interacción
- Necesidad de SEO
- Complejidad organizacional
- Proyección futura

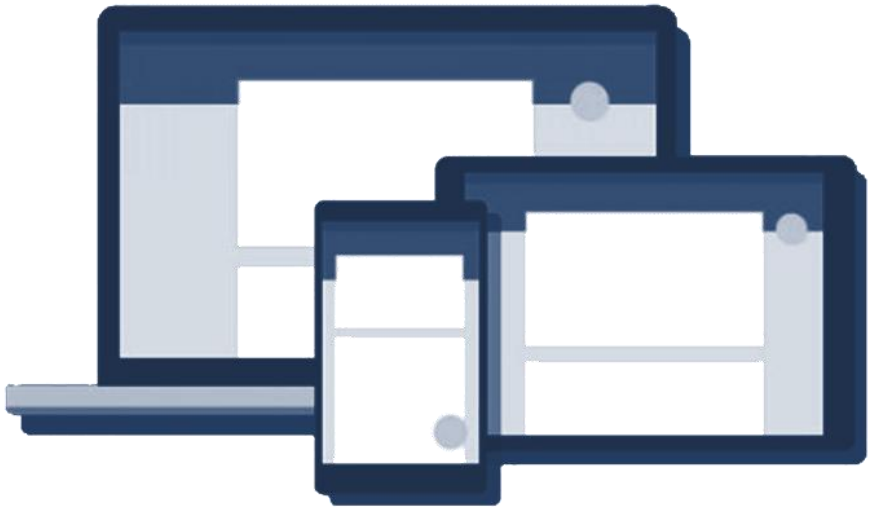
La decisión coherente es:

- SPA empresarial modular
- Arquitectura interna desacoplada
- Preparada para crecer

No es una decisión por tendencia.
Es una decisión arquitectónica fundamentada.

04 - Arquitecturas Frontend

¿Qué significa Arquitectura en Frontend?



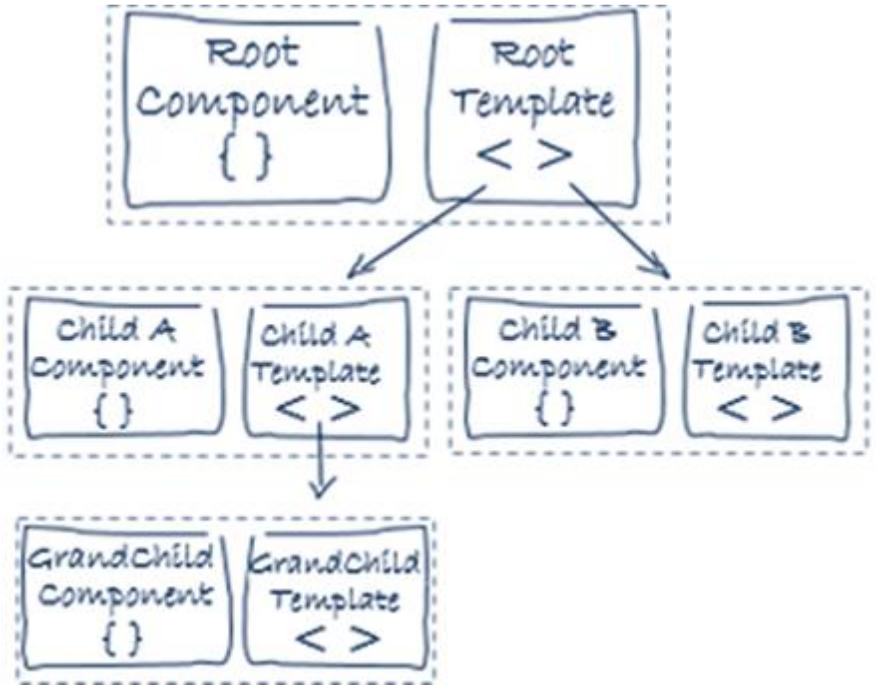
Arquitectura frontend es decidir:

- Dónde vive la lógica.
- Quién depende de quién.
- Cómo se organizan los dominios.
- Cómo evitamos acoplamiento futuro.

En Library, la pregunta concreta es:

¿Cómo evitamos que Authors, Books y Catalog se mezclen y se vuelvan un solo bloque de código?

Arquitectura por Capas Tradicional



Organización típica en Angular:

- Components
- Services
- Models
- API

Separación por tipo técnico, no por dominio.

Ventajas

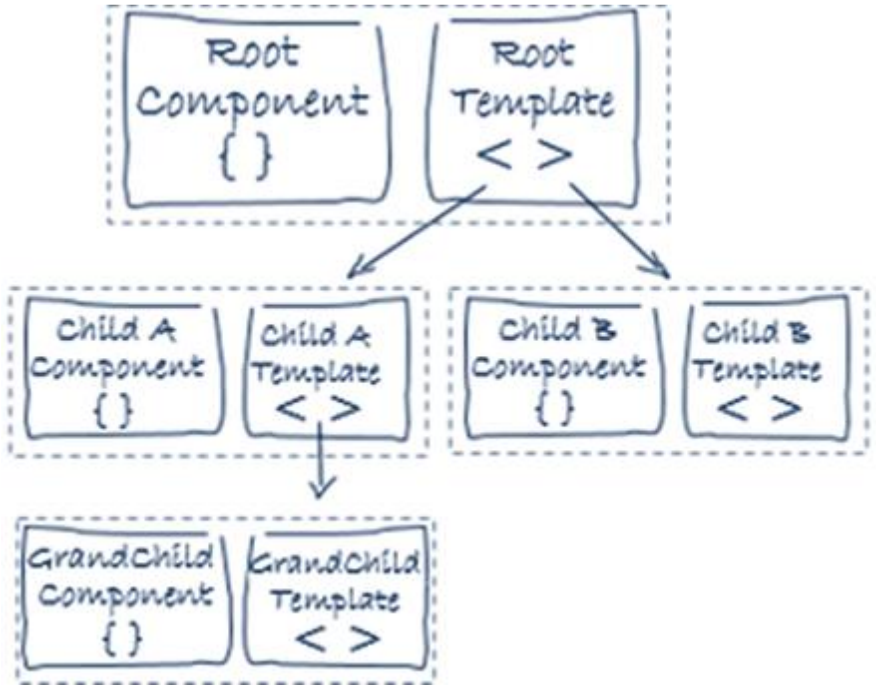
- Fácil de entender.
- Rápida de implementar.
- Funciona en proyectos pequeños.

Problemas cuando el sistema crece

- Services acumulan demasiada responsabilidad.
- Componentes empiezan a contener lógica.
- No hay límites claros entre dominios.
- Se generan dependencias cruzadas invisibles.

En sistemas empresariales esto escala mal.

Arquitectura por Capas Tradicional



Supongamos esta estructura:

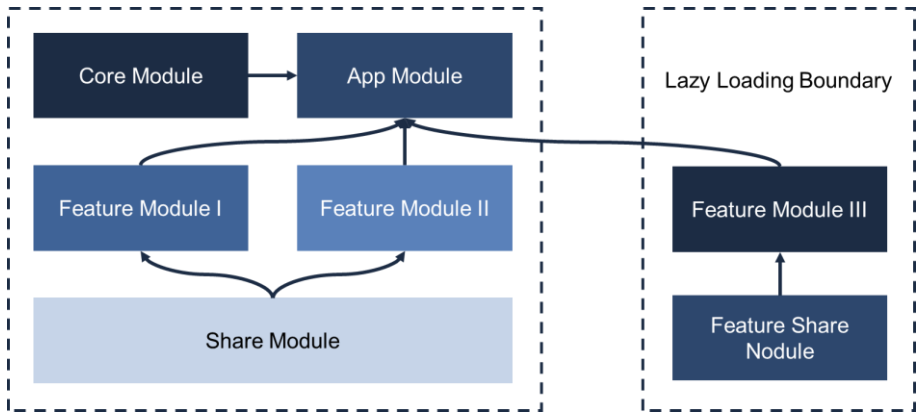
```
services/
   library.service.ts
components/
   authors.component.ts
   books.component.ts
   catalog.component.ts
```

library.service.ts:
- getAuthors()
- getBooks()
- getCatalog()
- mapea DTOs
- maneja errores
- aplica reglas
- mantiene estado

Problemas:
- Authors depende indirectamente de Books.
- Catalog puede reutilizar lógica de Authors.
- Todo está centralizado.
- Cambiar Books puede afectar Authors.

Esto rompe los límites del dominio.

Feature-First (Organización por Funcionalidad)



Organizar por funcionalidad:
- authors/
- books/
- catalog/

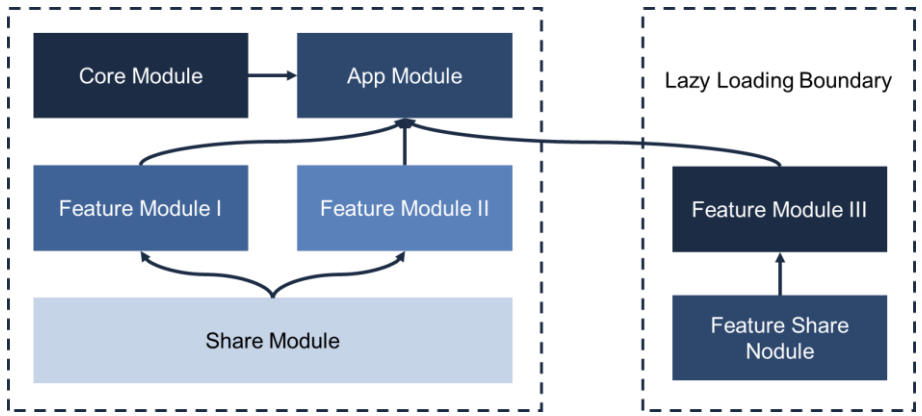
Cada carpeta contiene:
- Componentes
- Services
- Modelos

Ventajas
- Mejor separación funcional.
- Escala mejor que capas simples.
- Facilita trabajo por equipo.

Problemas
- Sigue mezclando responsabilidades internas.
- Service puede hacer HTTP + reglas + estado.
- No controla dirección de dependencias.

Es mejor, pero no suficiente para Library.

Feature-First (Organización por Funcionalidad)



Estructura:

```
authors/
    authors.component.ts
    authors.service.ts
    authors.model.ts
```

authors.service.ts:

- Llama HTTP
- Mapea DTO
- Valida datos
- Mantiene estado
- Decide reglas

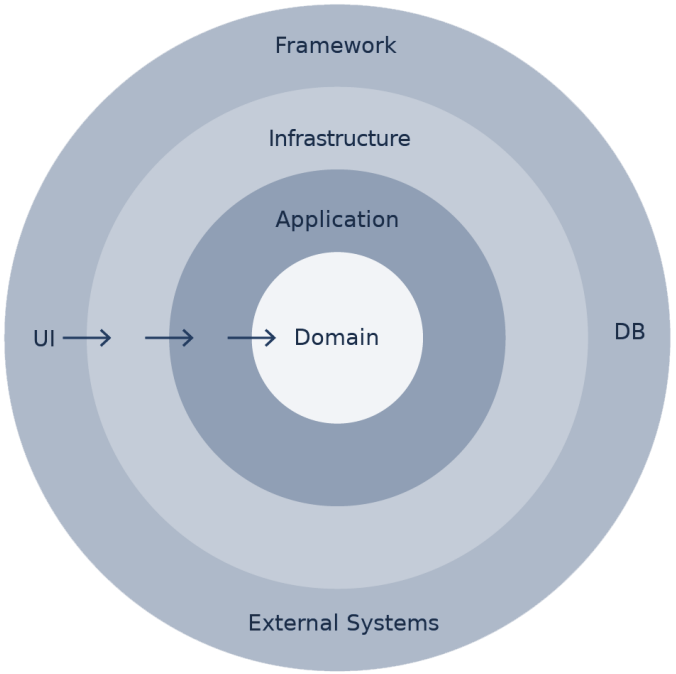
Problema:

El service sigue siendo una “mini capa backend” mezclada.

Si cambia el contrato REST, hay que modificar lógica y UI al mismo tiempo.

Separación visual ≠ separación arquitectónica.

DDD + Clean Architecture



Separación por dominio y por responsabilidad:
Capas conceptuales:

- Presentation
- Application
- Domain
- Infrastructure

Las dependencias apuntan hacia el dominio.

Ventajas

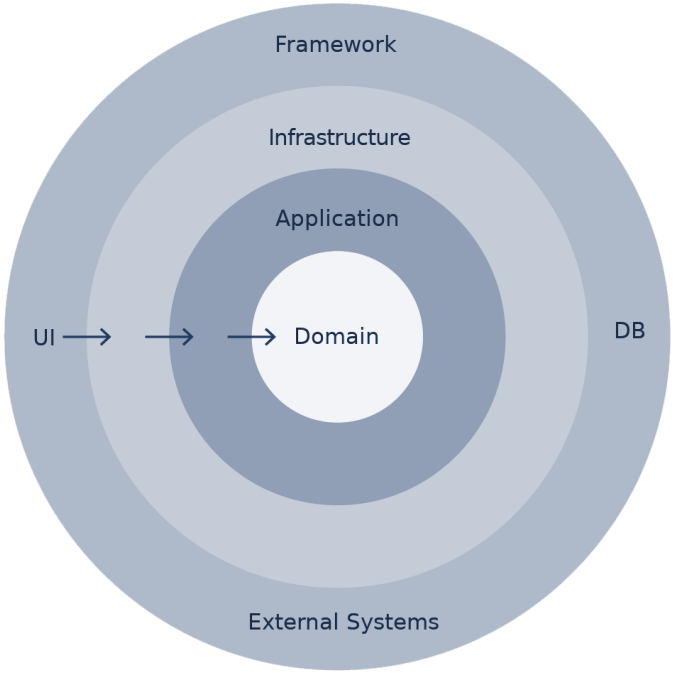
- Lógica fuera de la UI.
- HTTP aislado.
- DTOs no contaminan modelos internos.
- Testeo real sin backend.
- Control de dependencias.

Desafío

- Requiere disciplina.
- Más estructura inicial.
- Hay que entender responsabilidades.

Pero escala correctamente.

DDD + Clean Architecture



Dentro de Authors:

```
authors/
   presentation/
      authors-page.component.ts
application/
   get-authors.usecase.ts
domain/
   author.model.ts
infrastructure/
   authors-http.adapter.ts
```

Flujo real:

Component
→ GetAuthorsUseCase
→ AuthorsPort (interface)
→ AuthorsHttpAdapter
→ REST API

Beneficios concretos:

- El componente no conoce HTTP.
- Si cambia la API, solo cambia el adapter.
- Las reglas pueden testearse aisladas.
- Catalog no puede acceder a lógica interna de Authors.

Aquí sí respetamos el dominio.

05 - HISTORIA Y FILOSOFÍA DE ANGULAR

El Origen: AngularJS (2010)

Angular nace en 2010 como **AngularJS (v1.x)**, desarrollado en Google.

Contexto de la época

- Aplicaciones web principalmente multipágina.
- Mucha manipulación manual del DOM.
- JavaScript sin estructura clara.
- jQuery dominando el frontend.

Qué aportó AngularJS

- Two-way data binding.
- MVC en el navegador.
- Templates dinámicos.
- Separación básica de responsabilidades.

Fue revolucionario para construir las primeras SPA.

Pero tenía limitaciones:

- Difícil de escalar en proyectos grandes.
- Performance afectada en aplicaciones complejas.
- **Arquitectura poco estricta.**

La Ruptura: Angular 2 (2016)

En 2016 Google decide reescribir completamente el framework.

Angular 2 no fue una evolución menor, fue una reescritura total.

Cambios clave

- Basado en **TypeScript**.
- Arquitectura basada en componentes.
- Sistema robusto de Inyección de Dependencias.
- Mejor rendimiento.
- Compilador moderno.
- Soporte fuerte para testing.

AngularJS y Angular moderno no son la misma tecnología. Este cambio marcó el inicio del Angular que usamos hoy.

Evolución Continua (v2 - v21)

Desde 2016 Angular adopta:

- Versionado semestral.
- Roadmap público.
- Mejoras incrementales.

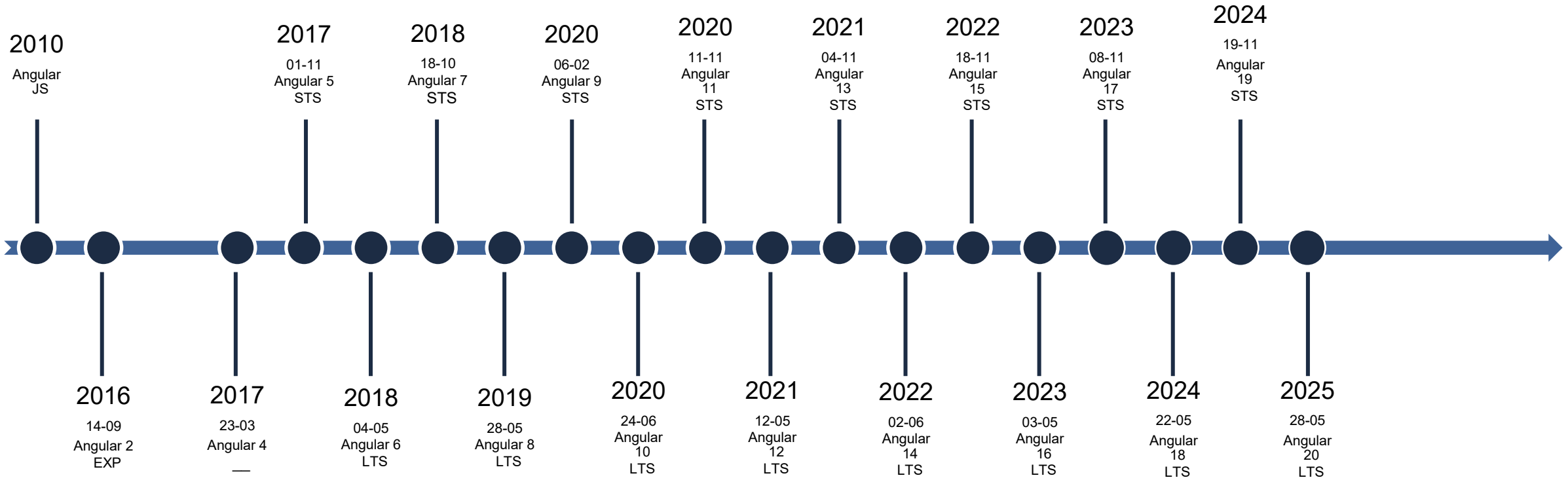
Evoluciones importantes en el tiempo:

- Ivy Renderer (optimización interna).
- Mejor tipado de templates.
- Standalone Components.
- Signals.
- Nuevo control flow.
- Optimización de builds.
- SSR moderno e hydration.

Angular no se quedó estático, ha evolucionado constantemente hacia menos boilerplate y mejor performance.

Historia y Filosofía de Angular

Versiones de Angular



Desde Angular **v4 en adelante**, Google definió un ciclo de **dos lanzamientos por año**:

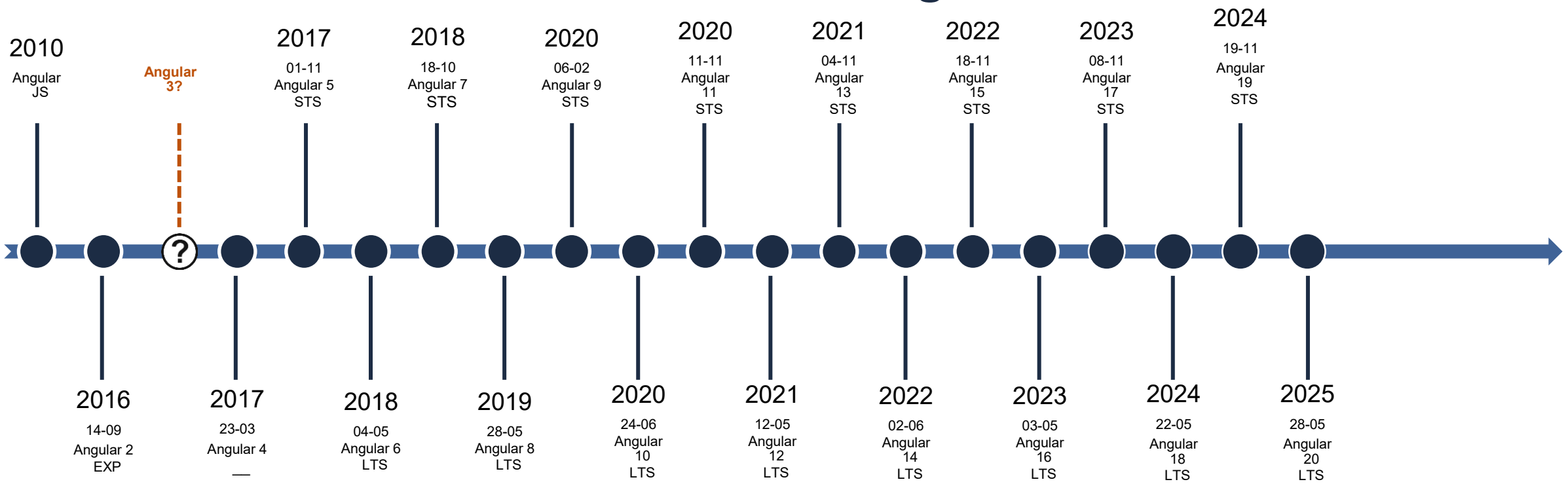
- **El primer release del año (mayo) es LTS**
- **El segundo release del año (noviembre) es STS**

AngularJS (v1) no seguía este ciclo formal

- **LTS** (Long Term Support) soporte de **18 meses**
- **STS** (Short Term Support) soporte de **6 meses**.

Historia y Filosofía de Angular

Versiones de Angular



Desde Angular **v4 en adelante**, Google definió un ciclo de **dos lanzamientos por año**:

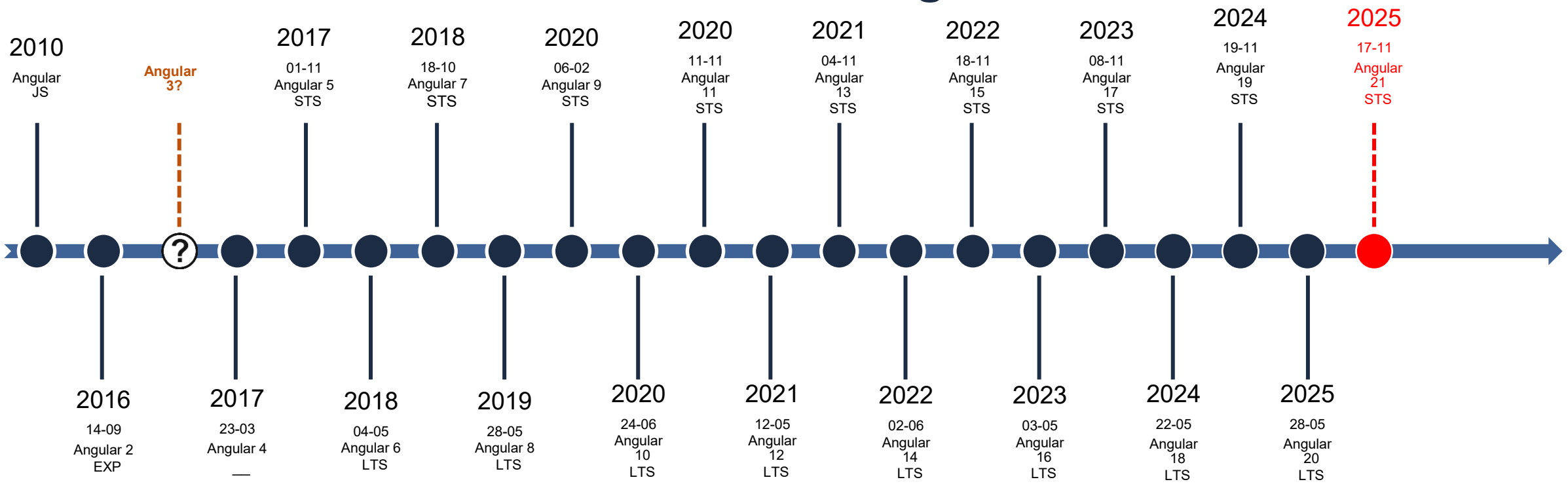
- **El primer release del año (mayo) es LTS**
- **El segundo release del año (noviembre) es STS**

AngularJS (v1) no seguía este ciclo formal

- **LTS** (Long Term Support) soporte de **18 meses**
- **STS** (Short Term Support) soporte de **6 meses**.

Historia y Filosofía de Angular

Versiones de Angular



Desde Angular **v4 en adelante**, Google definió un ciclo de **dos lanzamientos por año**:

- **El primer release del año (mayo) es LTS**
- **El segundo release del año (noviembre) es STS**

AngularJS (v1) no seguía este ciclo formal

- **LTS** (Long Term Support) soporte de **18 meses**
- **STS** (Short Term Support) soporte de **6 meses**.

Filosofía del Framework

Angular es un framework **opinionated**.

Eso significa:
- Define convenciones claras.
- Define estructura.
- Define herramientas oficiales.
- Reduce decisiones arbitrarias.

Ventajas de esta filosofía:
- Menos ambigüedad en equipos grandes.
- Mayor consistencia.
- Mejor mantenibilidad.
- Curva de aprendizaje estructurada.

No es minimalista.
Es estructurado.

¿Qué Propone Angular como enfoque?

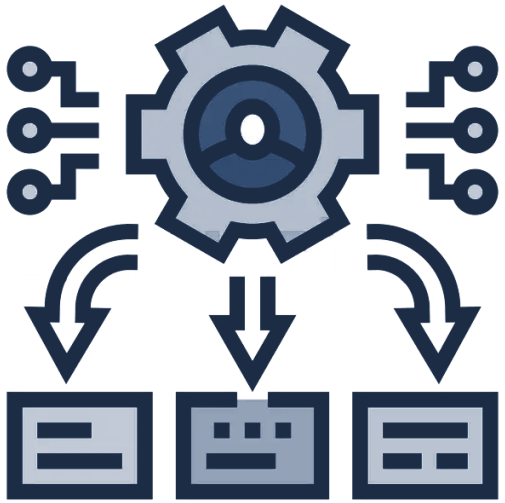
Angular propone:
- Aplicaciones estructuradas.
- Separación clara de responsabilidades.
- Uso fuerte de tipado.
- Inyección de dependencias.
- Testing como parte natural del desarrollo.

Es un framework diseñado para:
- Crecer.
- Mantenerse.
- Trabajar en equipo.

No es solo una herramienta para renderizar UI.
Es una plataforma completa para construir aplicaciones web modernas.

06 - CARACTERÍSTICAS TÉCNICAS PRINCIPALES DE ANGULAR

Angular como Framework Estructurado



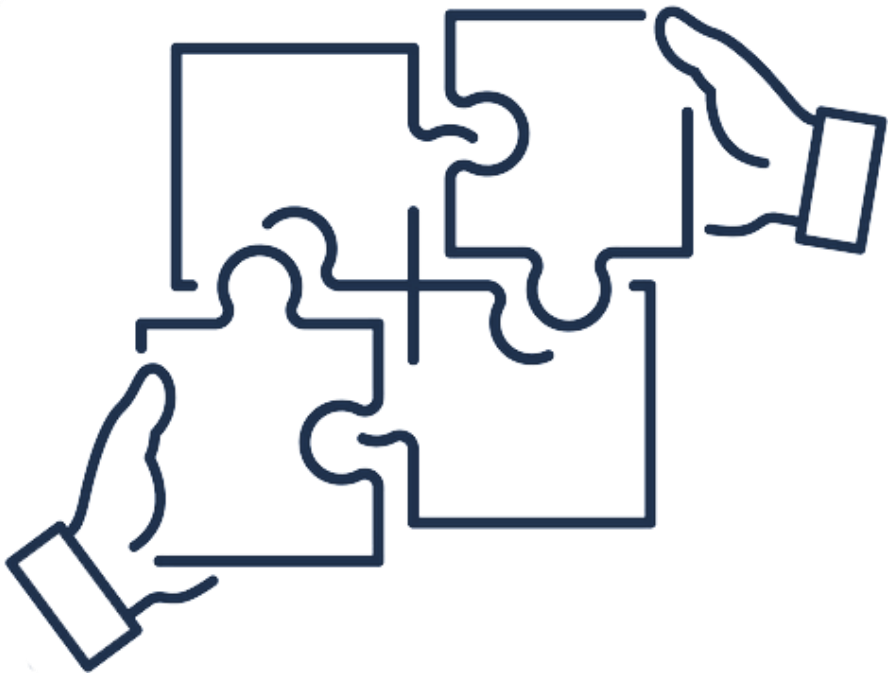
Angular propone una arquitectura basada en **componentes tipados**, donde cada pieza de la UI tiene:

- Template
- Lógica
- Estilos
- Metadatos claros

A diferencia de librerías ligeras, Angular impone estructura desde el inicio. El uso obligatorio de **TypeScript** introduce tipado fuerte, lo que permite modelar entidades como Author, Book o Catalog de forma segura y coherente con el backend.

Resultado: **Aplicaciones más predecibles, mantenibles y fáciles de escalar.**

Desacoplamiento y Reactividad



Angular integra de forma nativa dos capacidades clave:

Inyección de Dependencias (DI)
Permite desacoplar contratos de implementaciones. Un caso de uso puede depender de una interfaz, mientras Angular inyecta el adaptador concreto (por ejemplo, uno HTTP).

Esto facilita:
- Testing
- Sustitución de implementaciones
- Aplicación de Clean Architecture

Modelo Reactivo
Mediante Observables y Signals, Angular permite manejar asincronía y estado de forma estructurada. La UI reacciona automáticamente a cambios de datos sin manipulación manual del DOM.

Resultado: **Aplicaciones dinámicas sin perder control estructural.**

Escalabilidad y Ecosistema Integrado

Angular no es solo componentes.

Es una plataforma completa que incluye:

- Modularidad y lazy loading para dividir por dominios.
- CLI oficial para generar, construir y optimizar.
- Testing integrado desde el inicio.
- Herramientas de depuración oficiales.

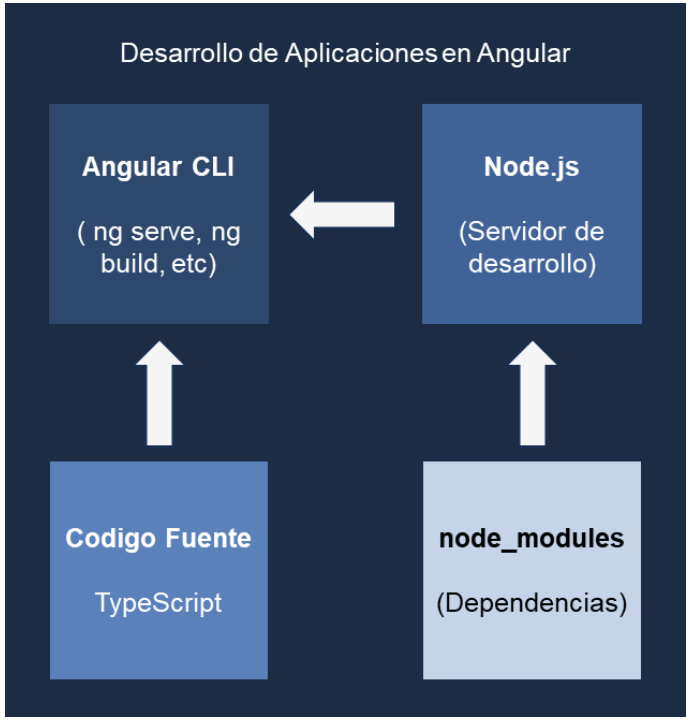
Esta combinación permite que una aplicación como Library:

- Crezca por módulos.
- Mantenga coherencia.
- Sea mantenible en el tiempo.
- No dependa de herramientas fragmentadas.

Angular reduce decisiones arbitrarias y estandariza la forma de construir aplicaciones.

07 – HERRAMIENTAS Y TOOLING PARA TRABAJAR CON ANGULAR

Entorno base de Desarrollo



Para desarrollar una aplicación Angular necesitamos:

Node.js (LTS)
Es el runtime que permite ejecutar el ecosistema de herramientas frontend.

npm (Node Package Manager)
- Gestión de dependencias.
- Instalación de librerías.
- Ejecución de scripts (build, serve, test).
- Control de versiones mediante package.json.

Angular depende completamente del ecosistema Node.

Entorno base de Desarrollo

Visual Studio Code (VS Code)
Editor recomendado por su integración con TypeScript y Angular.

Extensiones clave:
- Angular Language Service
- ESLint
- Prettier

Permite:
- Autocompletado avanzado.
- Validación de templates.
- Refactorizaciones seguras.
- Mejor experiencia de desarrollo.

Angular CLI

Es la herramienta oficial del framework.

Permite:

- Crear proyectos estructurados.
- Generar componentes y servicios.
- Ejecutar servidor de desarrollo.
- Compilar en modo producción.
- Ejecutar pruebas.

La CLI estandariza la estructura y evita configuraciones manuales complejas.

Ecosistema y Soporte Moderno

Reactividad

- RxJS para flujos complejos.
- Signals para estado simplificado en versiones modernas.

Linting y Calidad

- ESLint.
- Formateo automático.
- Buenas prácticas consistentes en equipo.

Ecosistema y Soporte Moderno

Integración y Escalabilidad

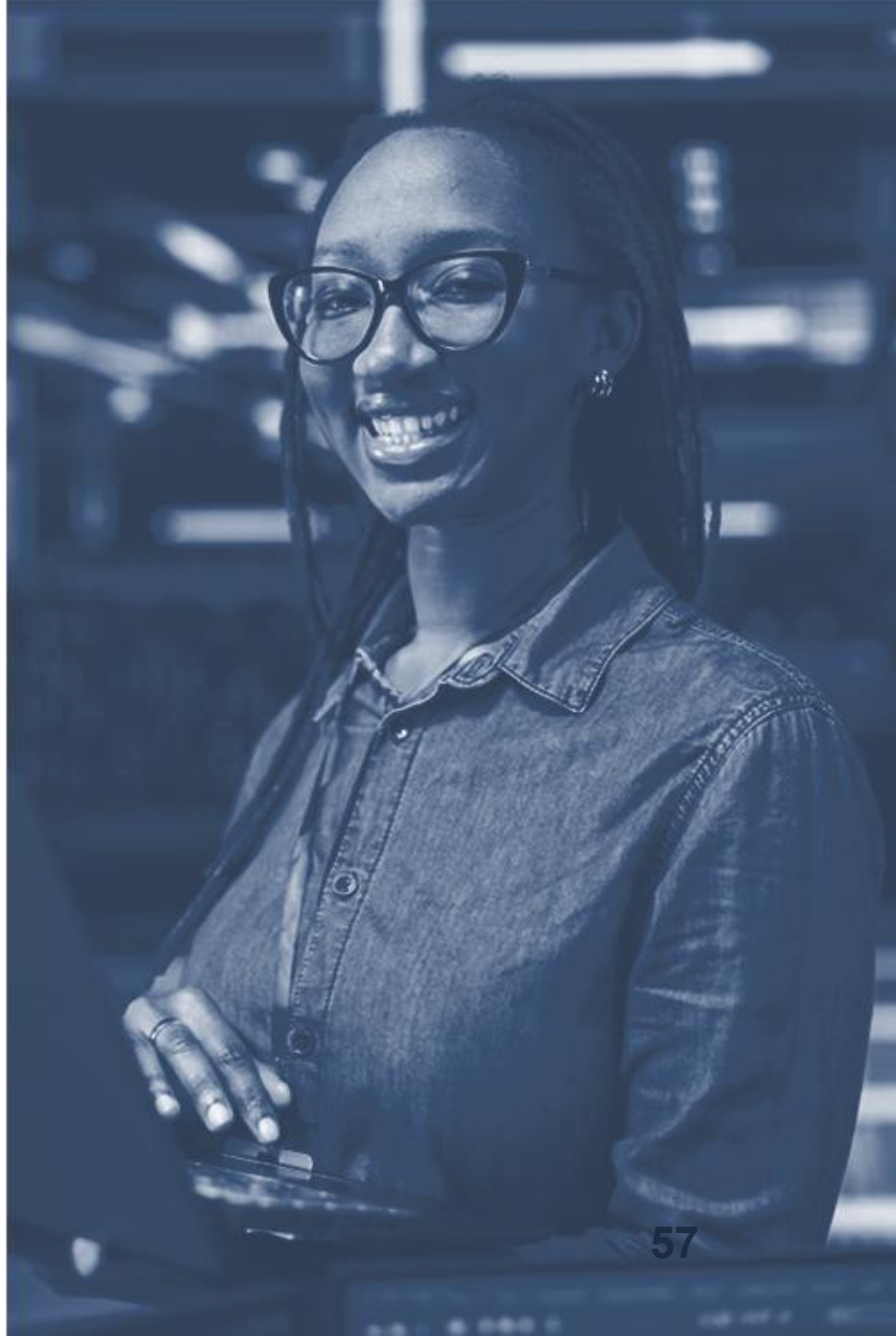
Angular se integra fácilmente con:

- Git para control de versiones.
- CI/CD pipelines.
- Docker.
- Monorepos (Nx).
- SSR cuando se requiere.

No es solo un framework de UI. es un entorno completo para construir aplicaciones modernas.

GRACIAS

POR SU PREFERENCIA



GALAXY
TRAINING

Ing. ERICK ARÓSTEGUI CUNZA