

{ MASTER DE TYPESCRIPT }

Guía de Arquitectura, Patrones y Escalabilidad para
Desarrolladores Modernos

LEWIS LÓPEZ GÓMEZ | Arquitecto de Software Senior



PRÓLOGO: CÓDIGO QUE SOBREVIVE

La mayoría de los desarrolladores iniciamos escribiendo código que simplemente "funciona": compila, pasa las pruebas básicas y cumple con lo mínimo para salir a producción. Pero cuando ese mismo código debe escalar, sobrevivir años, soportar miles de usuarios y convivir con múltiples equipos... la historia cambia.

Mi nombre es **Lewis Oswaldo López Gómez**, soy **Desarrollador FullStack y Arquitecto de Software** con más de una década construyendo soluciones para banca, educación y facturación electrónica. A lo largo de mi carrera, he trabajado con arquitecturas de microservicios y sistemas de misión crítica que no pueden "caerse".

En todos esos contextos, una constante se repite: **el tipado estático y las buenas prácticas marcan la diferencia entre un sistema frágil y uno profesional.**

TypeScript no es solo "**JavaScript con tipos**"; es una herramienta de arquitectura que te ayuda a diseñar mejores modelos de dominio y a prevenir errores antes de que lleguen a producción.

No encontrarás aquí una introducción "de juguete". Este libro está diseñado para que pienses como un profesional.

Bienvenido al Máster de TypeScript. Empecemos a escribir código que sea mantenable y escalable mañana.



Lewis Lopez G.

CÓMO USAR ESTE MANUAL

A. ¿PARA QUIÉN ES ESTE LIBRO?

1.  **Devs JavaScript:** Que buscan saltar al tipado estático profesional.
2.  **FullStack Modernos:** Que usan frameworks (NestJS, React) y quieren dominar la base.
3.  **Arquitectos:** Que buscan patrones sólidos y escalabilidad.

B. REGLAS DE JUEGO

- **Editor Abierto:** No solo leas. Copia, rompe y arregla los ejemplos.
- **Piensa en tu Proyecto:** Aplica cada patrón a tu API o sistema actual.
- **Toma Decisiones:** Anota cuándo usarás *interface* y cuándo *type* en tu equipo.
- **Mini Retos:** Al final de cada lección hay un reto. Hazlo. Es la única forma de interiorizar la teoría.

C. PRERREQUISITOS TÉCNICOS

Para sacarle el máximo provecho, idealmente deberías tener:

- **Node.js** instalado.
- Un editor moderno (**VS Code**, **WebStorm**, etc.) con soporte para **TypeScript**.
- Conocimientos básicos de:
 - Funciones, clases, módulos.
 - **Promesas y async/await.**
 - Uso de **npm/yarn**.

No necesitas saber ningún **framework** específico, pero si trabajas con **Angular**, **React** o **NestJS**, verás rápidamente cómo todo esto se conecta con tu día a día.

RECURSOS

-  **REPOSITORIO DE CÓDIGO** No tienes que copiar y pegar manualmente. Descarga todos los ejemplos, ejercicios y soluciones de los retos directamente desde el repositorio oficial: github.com/lopezsoft/master-typescript

HOJA DE RUTA: VOLUMEN 1

LECCIÓN 1.1 - ¿POR QUÉ TYPESCRIPT? La necesidad del tipado estático en entornos empresariales.

LECCIÓN 1.2 - ENTORNO PROFESIONAL Configuración avanzada de tsconfig.json para equipos.

LECCIÓN 1.3 - TIPOS FUNDAMENTALES Primitivos, Arrays y el peligro de any.

LECCIÓN 1.4 - INTERFACES VS TYPES La guía definitiva para modelar datos y contratos.

LECCIÓN 1.5 - POO Y CLASES Clases abstractas, modificadores y patrones de diseño reales.

LECCIÓN 1.6 - GENÉRICOS Creando componentes reutilizables y flexibles.

LECCIÓN 1.7 - DECORADORES La magia detrás de NestJS y la metaprogramación.

LECCIÓN 1.8 - ORGANIZACIÓN Y MÓDULOS Arquitectura de carpetas, Barrel files y Clean Code.

LECCIÓN 1.9 - ASINCRONÍA AVANZADA Promesas, Async/Await y el Event Loop.

1.1 ¿POR QUÉ TYPESCRIPT?

Más allá de la sintaxis: Seguridad y Escalabilidad.

TypeScript no es un lenguaje completamente nuevo que debas aprender desde cero. Es un superconjunto de JavaScript, lo que significa que todo código JavaScript válido es automáticamente código TypeScript válido. Sin embargo, TypeScript añade una capa fundamental que transforma la manera en que desarrollamos: el tipado estático.

En entornos profesionales, TypeScript es una herramienta de Gestión de Riesgos.

1.  **Tipado Estático:** Detecta errores en tiempo de compilación, mientras escribes código, en lugar de descubrirlos cuando el usuario ya está usando la aplicación. Esto reduce drásticamente los bugs en producción. En entornos profesionales, esto se traduce en menos tiempo de depuración y mayor confianza en el despliegue.
2.  **Predictibilidad:** El código se vuelve autodocumentado. Sabes exactamente qué tipo de datos entran y salen de cada función, mejorando la mantenibilidad del proyecto. Esto es vital en equipos grandes, donde múltiples desarrolladores trabajan en la misma base de código, facilitando el **onboarding** y la colaboración.
3.  **Potencia la IA:** Herramientas como **GitHub Copilot** funcionan significativamente mejor con TypeScript porque las interfaces estrictas proporcionan contexto claro sobre la estructura de datos. Esto optimiza la productividad del desarrollador al generar código más preciso y relevante, un activo clave en proyectos de gran escala.

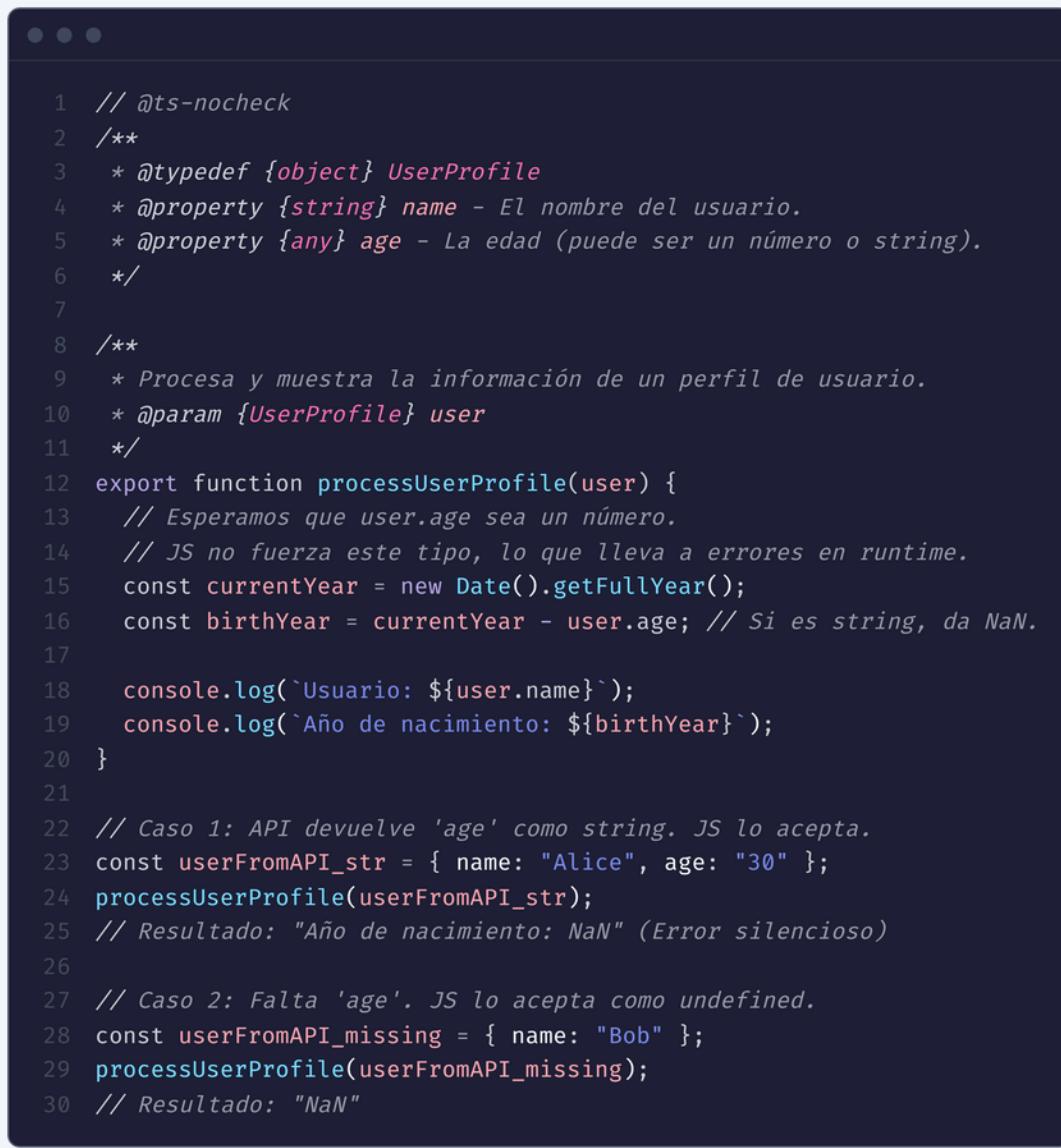
 **DATO DE LA INDUSTRIA** Un estudio de Microsoft reveló que el uso correcto de tipado estático puede prevenir hasta un **15% de los errores** que llegan a producción. En un sistema bancario o un SaaS masivo, ese 15% es la diferencia entre dormir tranquilo o apagar incendios un domingo.

EJEMPLO PRÁCTICO: GESTIÓN DE PERFILES

JavaScript (Riesgo de Errores Silenciosos)

En JavaScript, estos escenarios se ejecutan sin errores inmediatos, pero producen resultados lógicos incorrectos (`NaN`).

Estos problemas solo se descubren en tiempo de ejecución, a menudo por el usuario final o durante pruebas exhaustivas, lo que aumenta los costos de depuración y el riesgo de impacto negativo en la experiencia del usuario. En un entorno profesional, esto puede causar interrupciones y pérdidas de reputación.



```
1 // @ts-nocheck
2 /**
3  * @typedef {object} UserProfile
4  * @property {string} name - El nombre del usuario.
5  * @property {any} age - La edad (puede ser un número o string).
6 */
7
8 /**
9  * Procesa y muestra la información de un perfil de usuario.
10 * @param {UserProfile} user
11 */
12 export function processUserProfile(user) {
13     // Esperamos que user.age sea un número.
14     // JS no fuerza este tipo, lo que lleva a errores en runtime.
15     const currentYear = new Date().getFullYear();
16     const birthYear = currentYear - user.age; // Si es string, da NaN.
17
18     console.log(`Usuario: ${user.name}`);
19     console.log(`Año de nacimiento: ${birthYear}`);
20 }
21
22 // Caso 1: API devuelve 'age' como string. JS lo acepta.
23 const userFromAPI_str = { name: "Alice", age: "30" };
24 processUserProfile(userFromAPI_str);
25 // Resultado: "Año de nacimiento: NaN" (Error silencioso)
26
27 // Caso 2: Falta 'age'. JS lo acepta como undefined.
28 const userFromAPI_missing = { name: "Bob" };
29 processUserProfile(userFromAPI_missing);
30 // Resultado: "NaN"
```

Este código muestra cómo JSDoc no es suficiente y cómo falla la lógica en Runtime.

SOLUCIÓN (TYPESCRIPT): DETECCIÓN EN COMPILACIÓN

Contratos Fuertes y Seguridad

TypeScript, mediante la definición de la interfaz `UserProfile`, detecta inmediatamente que `age` debe ser un `number`. Cualquier intento de pasar un `string` o de omitir la propiedad requerida generará un error en tiempo de compilación.

Esto permite a los desarrolladores corregir el problema antes de que el código llegue a producción, garantizando la calidad de los datos y la robustez de la aplicación desde las primeras etapas.

```
1 // Definimos el contrato estricto
2 export interface UserProfile {
3   name: string;
4   age: number;      // La edad debe ser SIEMPRE un número
5   email?: string;  // Propiedad opcional
6   isActive?: boolean;
7 }
8
9 export function processUserProfile(user: UserProfile): void {
10  const currentYear: number = new Date().getFullYear();
11  const birthYear: number = currentYear - user.age;
12
13  console.log(`Usuario: ${user.name}`);
14  console.log(`Año de nacimiento: ${birthYear}`);
15 }
16
17 // --- VALIDACIÓN EN TIEMPO DE DISEÑO ---
18
19 // ✗ Error de compilación: Type 'string' is not assignable to 'number'.
20 // const userError: UserProfile = { name: "Alice", age: "30" };
21
22 // ✗ Error: Property 'age' is missing.
23 // const userMissing: UserProfile = { name: "Bob" };
24
25 // ✓ Correcto: Cumple la interfaz
26 const userOk: UserProfile = { name: "Charlie", age: 25, isActive: true };
27 processUserProfile(userOk);
```

 **DATO DE LA INDUSTRIA** Un estudio de Microsoft reveló que el uso de TypeScript puede prevenir hasta un **15% de los errores comunes** que llegan a producción. Empresas como **Airbnb** reportaron una **reducción del 80% en bugs** tras la migración.

1.2 CONFIGURACIÓN DEL ENTORNO

Un proyecto profesional empieza por su configuración. Muchos equipos dejan el archivo `tsconfig.json` por defecto por miedo a las alertas rojas.

En mis equipos, la regla es innegociable: **Si no es estricto, no es TypeScript.**

Activar el modo **strict: true** desde el día uno es la mejor inversión que puedes hacer. Aunque al principio exige más disciplina, te obliga a manejar los casos **null** y **undefined** explícitamente, eliminando la causa número uno de caídas en tiempo de ejecución (**runtime**).

Para establecer una base sólida, seguimos estos pasos estándar en cualquier proyecto empresarial:

1. **Iniciar Proyecto Node.js:** Ejecutar `npm init -y` para generar el `package.json`.
2. **Instalar TypeScript Localmente:** Usar `npm install typescript --save-dev` para garantizar que todo el equipo use la misma versión del compilador .

Generar Configuración Base: Ejecutar `npx tsc --init` como punto de partida.

```
1  {
2    "compilerOptions": {
3      "target": "es2020",           // Equilibrio entre modernidad y compatibilidad
4      "module": "commonjs",        // Estándar para backend Node.js
5      "lib": ["es2020", "dom"],    // Definiciones para ES2020 y DOM
6      "strict": true,             // ¡CRÍTICO! Habilita todas las verificaciones estrictas
7      "outDir": "./dist",         // Directorio de salida compilado
8      "rootDir": "./src",         // Directorio de código fuente
9      "esModuleInterop": true,    // Compatibilidad entre CommonJS y ES Modules
10     "skipLibCheck": true,       // Salta verificación de librerías (velocidad)
11     "forceConsistentCasingInFileNames": true, // Evita errores por mayúsculas/minúsculas
12     "moduleResolution": "node", // Algoritmo de resolución de Node.js
13     "baseUrl": "./src",        // Base para rutas absolutas
14     "paths": {                  // Alias para importaciones limpias
15       "@models/*": ["models/*"],
16       "@utils/*": ["utils/*"],
17       "@services/*": ["services/*"]
18     },
19     "sourceMap": true,          // Permite depurar TS en producción
20     "declaration": true,        // Genera archivos .d.ts (vital para librerías)
21     "noUnusedLocals": true,     // Error si hay variables sin usar (código limpio)
22     "noUnusedParameters": true, // Error si hay parámetros sin usar
23     "noEmitOnError": true      // No genera JS si hay errores de TS
24   },
25   "include": ["src/**/*"],      // Compilar todo en src
26   "exclude": ["node_modules", "dist"] // Ignorar dependencias y salida
27 }
```

Copia este JSON. Incluye todas las opciones y comentarios del original para que sea una referencia real.

EXPLICACIÓN DE OPCIONES CLAVE

Aquí explicamos el "por qué" de cada decisión técnica.

- **Target y Module:**
 - Usamos `es2020` para aprovechar características modernas y `commonjs` como estándar robusto para backends Node.js .
- **Strict (Estricto por Defecto):**
 - ¡La opción más crítica! Obliga a escribir código seguro. Reduce errores a largo plazo y facilita la colaboración al hacer las intenciones explícitas .
- **Organización (outDir / rootDir):**
 - Separar código fuente (`src`) y compilado (`dist`) es clave para los pipelines de CI/CD y despliegue limpio .
- **BaseUrl y Paths:**
 - Permite alias (ej. `@utils/logger`). Hace los `imports` legibles y simplifica la refactorización de carpetas en proyectos grandes .
- **SourceMap y Declaration:**
 - `sourceMap` es indispensable para depurar producción. `declaration` genera los archivos `.d.ts`, crucial si publicas librerías para otros equipos.

 **CONSEJO PROFESIONAL** Siempre mantén `"strict": true` activado desde el inicio. Migrar a modo estricto después de acumular código es costoso y doloroso. Es una inversión temprana en calidad .

RESUMEN Y RETO PRÁCTICO

Ideas Clave ✓

- `tsconfig.json` es el cerebro del compilador; defínelo bien.
- `strict: true` debe ser la norma en proyectos profesionales.
- Usa `baseUrl` y `paths` para evitar el infierno de rutas relativas (`../../../../`).
- Configurar TS no es "que compile", es alinear el compilador a los estándares de tu equipo.

Errores Comunes !

- Desactivar `strict` para "quitar errores rápido".
- Mezclar código fuente y compilado (no usar `dist`).
- Copiar un `tsconfig` de internet sin entender qué hace cada línea.

MINI RETO: CONFIGURACIÓN DESDE CERO 🎯

1. Crea un proyecto vacío con `npm init` y `tsc --init`.
2. Modifica el `tsconfig.json` para:
 - a. Activar `strict: true`.
 - b. Definir `rootDir: ./src` y `outDir: ./dist`.
 - c. Configurar `baseUrl` y un alias `@core/*` que apunte a `./src/core/*`.
3. Crea un archivo en `src/core/example.ts` e impórtalo en `src/app/main.ts` usando el alias.
4. Compila y verifica que funcione .

Este ejercicio te enseña a tratar el `tsconfig.json` como parte esencial del diseño de tu proyecto, no como un archivo "que se deje por defecto".

LECCIÓN 1.3: TIPOS DE DATOS FUNDAMENTALES

TypeScript extiende significativamente el sistema de tipos de JavaScript, proporcionando herramientas poderosas para describir con precisión la forma de tus datos. Dominar estos tipos fundamentales es esencial para escribir código TypeScript robusto y mantenible en un entorno profesional.

En el desarrollo profesional, los tipos primitivos (**string, number, boolean**) son la base para construir modelos de datos complejos. Al declararlos explícitamente, especialmente dentro de interfaces o clases, defines contratos de datos claros que son críticos para la consistencia.

El tipo any: Es una "válvula de escape" que desactiva las verificaciones de tipo. Su uso debe ser minimizado y justificado (ej. al recibir JSON de APIs externas). La mejor práctica es validar y re-tipar estos datos lo antes posible.

```
1 // user.model.ts
2 export interface SystemUser {
3   id: string; // Identificador único del usuario.
4   fullName: string; // Nombre completo y apellido.
5   emailVerified: boolean; // Indica si el correo ha sido verificado.
6   assignedRoles: string[]; // Lista de roles (ej. "admin").
7   permissions: Array<number>; // IDs numéricos de permisos.
8 }
9
10 // app.ts
11 let currentUser: SystemUser = {
12   id: "USR-001-XYZ",
13   fullName: "Ana García",
14   emailVerified: true,
15   assignedRoles: ["admin", "developer", "tester"],
16   permissions: [101, 203, 400]
17 };
18
19 // api.service.ts (Uso de 'any')
20 let genericAPIData: any = {
21   message: "Carga exitosa",
22   payload: [ { itemId: 1, name: "Producto A" } ]
23 };
24
25 // Es crucial validar los tipos cuando se trabaja con 'any'
26 if (typeof genericAPIData.message === "string") {
27   console.log(`Mensaje de API: ${genericAPIData.message}`);
28 }
```

TUPLAS Y ENUMS: DATOS ESTRUCTURADOS

Tuplas: Son arrays de longitud fija donde el tipo de dato de cada elemento en una posición específica está predefinido. Son excelentes para representar conjuntos de datos relacionados con una estructura conocida (ej. retornar múltiples valores).

Enums: Son conjuntos de constantes nombradas, haciendo el código más legible y menos propenso a errores de "cadenas mágicas". Son perfectos para estados, roles, niveles de severidad o tipos de eventos.

```
1 // user.model.ts
2 export interface SystemUser {
3   id: string; // Identificador único del usuario.
4   fullName: string; // Nombre completo y apellido.
5   emailVerified: boolean; // Indica si el correo ha sido verificado.
6   assignedRoles: string[]; // Lista de roles (ej. "admin").
7   permissions: Array<number>; // IDs numéricos // 1. TUPLAS (operation.types.ts)
8 // [éxito, mensaje, datos opcionales]
9 export type OperationResult = [boolean, string, any?];
10
11 export function processRequest(id: string): OperationResult {
12   if (id === "INVALIDO") {
13     return [false, "ID de solicitud no válido"];
14   }
15   return [true, "Solicitud procesada con éxito", { data: id }];
16 }
17
18 // 2. ENUMS (constants.ts)
19 export enum HttpStatusCode {
20   OK = 200,
21   CREATED = 201,
22   UNAUTHORIZED = 401,
23   SERVER_ERROR = 500
24 }
25
26 export enum NotificationType {
27   INFO = "information",
28   WARNING = "warning",
29   ERROR = "error",
30   SUCCESS = "success"
31 }de permisos.
32 }
33
34 // app.ts
35 let currentUser: SystemUser = {
36   id: "USR-001-XYZ",
37   fullName: "Ana Garcia",
38   emailVerified: true,
39   assignedRoles: ["admin", "developer", "tester"],
40   permissions: [101, 203, 400]
41 };
42
43 // api.service.ts (Uso de 'any')
44 let genericAPIData: any = {
45   message: "Carga exitosa",
46   payload: [ { itemId: 1, name: "Producto A" } ]
47 };
48
49 // Es crucial validar los tipos cuando se trabaja con 'any'
50 if (typeof genericAPIData.message === "string") {
51   console.log(`Mensaje de API: ${genericAPIData.message}`);
52 }
```

💡 CONSEJO: Las tuplas son increíblemente útiles cuando una función necesita retornar múltiples valores relacionados con tipos heterogéneos. Los Enums encapsulan opciones, permitiendo una refactorización más segura.

RESUMEN Y RETO PRÁCTICO

Ideas Clave ✓

- TypeScript extiende los tipos primitivos con un sistema más expresivo y seguro.
- Los **arrays** y colecciones se pueden tipar de forma explícita (**string[]**, **Array<number>**).
- **any** es el tipo "*escapatoria*": desactiva el sistema de tipos y debería usarse solo en casos muy puntuales.
- Modelar bien los tipos desde el inicio simplifica la evolución del sistema.

Errores Comunes !

- Declarar propiedades o respuestas de API como **any** por comodidad.
- Mezclar tipos de manera inconsistente dentro de una misma colección.
- No aprovechar tipos utilitarios como **Omit** o **Partial** cuando el dominio lo pide.

MINI RETO: MODELADO DE ENTIDAD

1. Piensa en una entidad real de tu contexto (**User**, **Order**, **Payment**).
2. Define:
 - a. Un tipo o interfaz para la entidad guardada en base de datos (**con id, fechas, flags**).
 - b. Un tipo separado para la creación de la entidad (usando **Omit** para excluir campos autogenerados).
3. Reemplaza en un ejemplo todos los usos de **any** relacionados con esa entidad por tus tipos concretos.

LECCIÓN 1.4: INTERFACES VS TYPES

Modelado de Datos Extensible

Una de las preguntas más frecuentes es: ¿cuándo debo usar **interface** y cuándo **type**?

Aunque hay superposición, cada uno tiene casos de uso óptimos.

Interfaces: Diseñadas específicamente para definir la forma de objetos y contratos de clases. Soportan "**declaration merging**" (pueden **redeclararse** para añadir campos). Son ideales para modelos de datos extensibles.

```
1 // src/types/product.ts
2
3 /**
4  * Define la estructura base de un producto.
5  * Ideal para modelos extensibles y contratos de API.
6 */
7 export interface IProduct {
8     readonly id: string; // Inmutable (UUID)
9     name: string;
10    description?: string; // Opcional
11    price: number;
12    stock: number;
13    creationDate: Date;
14    tags: string[];
15    supplier: {
16        id: string;
17        name: string;
18        contactEmail?: string;
19    };
20    // Método opcional en el contrato
21    calculateFinalPrice?(tax: number): number;
22 }
23
24 /**
25  * Extiende IProduct para productos en promoción.
26  * Añade campos específicos de descuento.
27 */
28 export interface IPromotionalProduct extends IProduct {
29     discountPercentage: number;
30     promotionEndDate: Date;
31     promotionCode: string;
32 }
```

TYPES: FLEXIBILIDAD Y COMPOSICIÓN

Types: Más flexibles y versátiles. Pueden definir primitivos, uniones, intersecciones y tuplas. No soportan declaration merging.

En este ejemplo profesional, usamos **types** para crear identificadores, estados fijos y combinar tipos existentes (Intersección e utilitarios como Pick).

```
1 // src/types/utility.ts
2 // Alias para IDs que pueden ser numéricos o strings
3 export type UniqueID = number | string;
4
5 // Union Literal Type para estados fijos (Evita magic strings)
6 export type TransactionStatus =
7   | "pending"
8   | "processing"
9   | "completed"
10  | "failed";
11
12 // src/types/api.ts
13 import { IProduct, IUser } from "./interfaces";
14
15 /**
16 * Intersección (&): Combina Product con detalles de inventario.
17 * Útil para reportes ad-hoc.
18 */
19 export type ProductWithInventoryDetail = IProduct & {
20   warehouseLocation: string;
21   lastInventoryUpdate: Date;
22   // Utility Type (Pick): Selecciona solo id y fullName de User
23   auditor: Pick<IUser, "id" | "fullName">;
24 };
25
26 /**
27 * Generico para estandarizar respuestas API.
28 */
29 export type ApiResponse<T> = {
30   data: T;
31   message: string;
32   status: "success" | "error";
33   timestamp: string;
34 };
```

USO REAL EN SERVICIOS

Veamos cómo se consumen estas interfaces y tipos en un servicio real. Observa cómo ApiResponse envuelve a IProduct o IPromotionalProduct, garantizando un tipado estricto en el retorno de las funciones asíncronas.

```
1 // src/services/productService.ts
2 import { IProduct, IPromotionalProduct } from "../types/product";
3 import { ApiResponse } from "../types/api";
4 import { UniqueID } from "../types/utility";
5
6 export async function getProductById(
7     productId: UniqueID,
8 ): Promise<ApiResponse<IProduct>> {
9     // Simulación de llamada a API
10    const response: ApiResponse<IProduct> = {
11        data: {
12            id: "PROD-001",
13            name: "Laptop Gamer X",
14            price: 1200.5,
15            stock: 50,
16            creationDate: new Date(),
17            tags: ["electronics", "gaming"],
18            supplier: { id: "SUP-ABC", name: "Tech Suppliers" },
19        },
20        message: "Producto obtenido exitosamente.",
21        status: "success",
22        timestamp: new Date().toISOString(),
23    };
24    return Promise.resolve(response);
25 }
26
27 export async function createPromotionalProduct(
28     data: IPromotionalProduct,
29 ): Promise<ApiResponse<IPromotionalProduct>> {
30     // Lógica de creación...
31     return Promise.resolve({
32         data: data,
33         message: "Producto promocional creado.",
34         status: "success",
35         timestamp: new Date().toISOString(),
36     });
37 }
```

GUÍA DE DECISIÓN Y RETO

Cuándo usar cada uno

- **Usa Interface:**
 - Definir la forma de objetos de datos (**DTOs**).
 - Contratos para clases (**implements**).
 - Definir **APIs** públicas en librerías (por el declaration merging).
 - Jerarquía de tipos (herencia **OOP**).
- **Usa Type:**
 - Crear Union Types (A | B).
 - Crear Intersection Types (A & B).
 - Tipos literales o Alias de primitivos.
 - Transformaciones avanzadas (**Pick**, **Omit**, **Partial**).

 **REGLA DE ORO:** Si tu objetivo principal es describir la forma de un objeto o definir un contrato extensible, **usa interface**. Si necesitas crear un alias, combinar tipos complejos o usar utilitarios, **usa type**.

MINI RETO: ARQUITECTURA DE TIPOS



1. Toma una entidad de tu proyecto (**Usuario**, **Producto**, **Pedido**) y define:
 - a. Una **interfaz base** con las propiedades principales.
 - b. Una **interfaz extendida** que herede de la base.
 - c. Un **type** que combine la interfaz base con info adicional usando intersección (&).
2. Crea un **type de unión** para los estados de esa entidad (ej. "active" | "archived").
3. Define un genérico **ApiResponse<T>** y úsallo en una función.

LECCIÓN 1.5: CLASES Y POO PROFESIONAL

Diseño de Arquitecturas Sólidas

TypeScript eleva la Programación Orientada a Objetos (**OOP**) de JavaScript a un nivel profesional, añadiendo características como *modificadores de acceso*, *clases abstractas* e *interfaces*. Estas herramientas te permiten diseñar *arquitecturas de software* mantenibles y escalables.

Ejemplo Profesional: Imaginemos un sistema de pagos que debe interactuar con diferentes pasarelas (**Tarjeta**, **PayPal**, **Cripto**). La **OOP** nos permite definir un contrato común y manejar las particularidades de cada una de forma modular.

```
1 // interfaces/PaymentGateway.ts
2
3 /**
4  * Define el contrato para cualquier pasarela de pago.
5  * Asegura que todas implementen métodos esenciales.
6 */
7 export interface PaymentGateway {
8     getTransactionId(): string;
9
10    processPayment(amount: number): boolean;
11
12    getStatus(): 'PENDING' | 'COMPLETED' | 'FAILED';
13
14    refundPayment(transactionId: string, amount: number): boolean;
15 }
```

CLASES ABSTRACTAS Y REUTILIZACIÓN

Clase Abstracta (BaseProcessor): Sirve como base para todas las pasarelas. Implementa funcionalidades comunes (como generar IDs) y define métodos abstractos que las hijas deben implementar.

Modificadores de Acceso:

- **protected:** Accesible solo en la clase base y sus hijas (ideal para lógica interna compartida).
- **public:** Accesible desde cualquier lugar (**API** pública).
- **abstract:** Obliga a la subclase a proveer la implementación.

```
 1 // baseProcessors/BaseProcessor.ts
 2 import { PaymentGateway } from "../interfaces/PaymentGateway";
 3
 4 export abstract class BaseProcessor implements PaymentGateway {
 5   protected transactionId: string;
 6   protected status: "PENDING" | "COMPLETED" | "FAILED" = "PENDING";
 7
 8   constructor() {
 9     this.transactionId = this.generateUniqueId();
10   }
11
12   public getTransactionId(): string {
13     return this.transactionId;
14   }
15
16   public getStatus(): "PENDING" | "COMPLETED" | "FAILED" {
17     return this.status;
18   }
19
20   // Métodos abstractos (El contrato forzoso para las hijas)
21   public abstract processPayment(amount: number): boolean;
22   public abstract refundPayment(transactionId: string, amount: number): boolean;
23
24   private generateUniqueId(): string {
25     return Math.random().toString(36).substring(2).toUpperCase();
26   }
27 }
```

IMPLEMENTACIÓN: PROCESADOR DE TARJETAS

Aquí vemos la **Herencia** (`extends`) en acción. `CardProcessor` extiende de `BaseProcessor`.

Observa cómo esta clase concreta **NO** se preocupa por generar el ID de transacción (eso lo hace el padre), sino que se enfoca exclusivamente en la lógica de negocio específica de las tarjetas: tokenización y validación.

```
1 // processors/CardProcessor.ts
2 import { BaseProcessor } from "../baseProcessors/BaseProcessor";
3
4 export class CardProcessor extends BaseProcessor {
5     private tokenizedCardNumber: string;
6     private expirationDate: string;
7
8     constructor(cardToken: string, expirationDate: string) {
9         super(); // Llama al constructor del parent (genera el ID)
10        this.tokenizedCardNumber = cardToken;
11        this.expirationDate = expirationDate;
12    }
13
14    public processPayment(amount: number): boolean {
15        console.log(
16            `Procesando ${amount} con tarjeta ${this.tokenizedCardNumber}`,
17        );
18
19        // Simular llamada a API externa (Stripe, Adyen)
20        if (Math.random() > 0.1) {
21            this.status = "COMPLETED"; // Accede a propiedad protected
22            return true;
23        }
24        this.status = "FAILED";
25        return false;
26    }
27
28    public refundPayment(transactionId: string, amount: number): boolean {
29        console.log(`Reembolsando ${amount} para ID: ${transactionId}`);
30        return true;
31    }
32 }
```

POLIMORFISMO EN ACCIÓN

El verdadero poder de la OOP: **El Polimorfismo**.

La función `executePayment` no sabe si está procesando una tarjeta o PayPal. Solo sabe que recibe algo que cumple la interfaz `PaymentGateway`. Esto permite añadir nuevos métodos de pago sin tocar el código del servicio principal (**Principio Open/Closed**).

```
● ● ●

1 // services/paymentService.ts
2 import { PaymentGateway } from "../interfaces/PaymentGateway";
3
4 // Recibe la Interfaz, no la clase concreta
5 export function executePayment(
6   gateway: PaymentGateway,
7   amount: number,
8 ): string {
9   if (gateway.processPayment(amount)) {
10     return `Pago exitoso. ID: ${gateway.getTransactionId()}`;
11   } else {
12     return `Pago fallido. Estado: ${gateway.getStatus()}`;
13   }
14 }
15
16 // --- APP.TS (USO) ---
17 import { CardProcessor } from "../processors/CardProcessor";
18 import { PayPalProcessor } from "../processors/PayPalProcessor";
19
20 const cardGateway = new CardProcessor("TOK_123", "12/25");
21 const paypalGateway = new PayPalProcessor("user@email.com");
22
23 // El servicio funciona igual para ambos
24 console.log(executePayment(cardGateway, 100));
25 console.log(executePayment(paypalGateway, 50));
```

RESUMEN Y RETO DE ARQUITECTURA

Resumen

- **Modificadores:** `public`, `private` y `protected` controlan el encapsulamiento.
- **Clases Abstractas:** Definen comportamientos base para reutilización.
- **Interfaces:** Garantizan contratos que facilitan el intercambio de implementaciones.
- **Polimorfismo:** Permite escribir código desacoplado que trabaja con abstracciones, no con concreciones.

Errores Comunes:

- Hacer todo `public` por defecto (pierdes control).
- Crear herencias demasiado profundas (**complejidad innecesaria**).
- **No usar interfaces** para definir contratos entre módulos.

💡 **LA REGLA DE ORO: PRINCIPIO "OPEN/CLOSED"**: Tu arquitectura debe estar abierta a la extensión pero cerrada a la modificación.

Gracias al **Polimorfismo** que acabamos de implementar, el día de mañana podrías agregar una nueva pasarela (ej. `BitcoinProcessor`) sin tener que tocar ni una sola línea del código del `PaymentService` que ya funciona en producción. Eso es escalabilidad real.

MINI RETO: SISTEMA DE NOTIFICACIONES



1. Diseña un sistema de notificaciones.
2. Crea una interfaz `INotifier` con métodos `send(message)` y `getStatus()`.
3. Implementa una clase abstracta `BaseNotifier` con lógica común (`logging`).
4. Crea clases concretas `EmailNotifier` y `SMSNotifier` que extiendan la base.
5. Escribe una función que reciba `INotifier` y úsala con ambas.

LECCIÓN 1.6: GENÉRICOS - REUTILIZACIÓN

Componentes Flexibles y Seguros

Los genéricos son la alternativa profesional al uso de `any`. Te permiten crear componentes que funcionan con múltiples tipos de datos, manteniendo la seguridad de tipos completa.

Ejemplo Profesional: En el desarrollo de **APIs**, es fundamental tener un formato de respuesta consistente (`ApiResponse`) que maneje tanto datos exitosos como errores de manera *tipada*.

```
1 // utils/apiResponse.ts
2
3 export interface ApiResponse<T> {
4     success: boolean;
5     message: string;
6     data: T | null; // El payload es genérico
7     timestamp: string;
8     statusCode: number;
9 }
10
11 export function createApiResponse<T>(
12     success: boolean,
13     data: T | null,
14     message: string,
15     statusCode: number = 200,
16 ): ApiResponse<T> {
17     return {
18         success,
19         data,
20         message,
21         timestamp: new Date().toISOString(),
22         statusCode,
23     };
24 }
```

GENÉRICOS EN ACCIÓN

Veamos cómo consumimos este **wrapper** genérico. Observa que no necesitamos crear una interfaz diferente para **UserResponse** o **ProductResponse**.

Reutilizamos la misma estructura, pero **TypeScript** sabe exactamente qué hay dentro de data.

```
1 // models/user.ts
2 interface User {
3   id: string;
4   name: string;
5 }
6
7 // models/product.ts
8 interface Product {
9   sku: string;
10  price: number;
11 }
12
13 // --- USO EN CONTROLADORES ---
14
15 const user: User = { id: "1", name: "Alice" };
16 // TypeScript infiere: ApiResponse<User>
17 const userResp = createApiResponse(true, user, "Usuario encontrado");
18
19 const products: Product[] = [{ sku: "P1", price: 100 }];
20 // TypeScript infiere: ApiResponse<Product[]>
21 const prodResp = createApiResponse(true, products, "Lista cargada");
22
23 // Acceso seguro (Autocompletado funciona)
24 console.log(userResp.data?.name);
25 console.log(prodResp.data?.[0].price);
```

LECCIÓN 1.7: DECORADORES

Los decoradores son funciones especiales que pueden modificar clases, métodos o propiedades. Son fundamentales en frameworks como **NestJS** y **Angular** para la **Inyección de Dependencias**.

Nota: Requieren "experimentalDecorators": true en **tsconfig.json**.

```
1 // decorators/Registrable.ts
2
3 // Decorador de Clase
4 function RegistrableService(serviceName: string) {
5     return function (constructor: Function) {
6         console.log(`[IoC] Registrando servicio: ${serviceName}`);
7         // Aquí se añadiría al contenedor de dependencias
8         Object.defineProperty(constructor.prototype, "_serviceName", {
9             value: serviceName,
10            writable: false,
11        });
12    };
13 }
14
15 @RegistrableService("AuthService")
16 export class AuthService {
17     login() {
18         console.log("Logueando...");
19     }
20 }
```

DECORADORES EN UN API REST

Este ejemplo muestra cómo **NestJS** utiliza decoradores para construir un controlador **RESTful** declarativo.

En lugar de configurar rutas manualmente, usamos **@Get**, **@Post** y **@Body** para decirle al **framework** qué hacer.

```
1 // controllers/users.controller.ts
2 import { Controller, Get, Post, Body, Param } from "@nestjs/common";
3 import { CreateUserDto } from "../dto/user.dto";
4
5 @Controller("api/v1/users")
6 export class UsersController {
7   constructor(private readonly userService: UserService) {}
8
9   @Get()
10  getAllUsers() {
11    return this.userService.findAll();
12  }
13
14  @Get(":id")
15  getUserId(@Param("id") id: string) {
16    return this.userService.findOne(id);
17  }
18
19  @Post()
20  createUser(@Body() createUserDto: CreateUserDto) {
21    // NestJS valida el DTO automáticamente antes de llegar aquí
22    return this.userService.create(createUserDto);
23  }
24 }
```

RESUMEN Y RETOS AVANZADOS

Ideas Clave

- **Genéricos:** Permiten crear código flexible que mantiene el tipado (*como Array<T> o Promise<T>*).
- **Decoradores:** Permiten programación *declarativa* y *metaprogramación*. Son la base de la Inversión de Control en **NestJS**.

 **LA TRAMPA DE LA "MAGIA":** Los decoradores son poderosos porque ocultan complejidad, pero eso tiene un precio: hacen que el flujo del código sea menos evidente.

Mi Regla: Úsalos estrictamente para '*Cross-Cutting Concerns*' (*cosas transversales como Logging, Caching, Validación o Seguridad*). Nunca escondas lógica de negocio crítica dentro de un decorador, o tu equipo sufrirá tratando de entender por qué ocurren las cosas.

MINI RETO: CACHÉ GENÉRICA



Crea una clase **Cache<T>** con métodos:

- `set(key: string, value: T): void`
- `get(key: string): T | undefined` Úsala para guardar User y Product sin perder el tipado.

MINI RETO: LOGGING AUTOMÁTICO



Implementa un decorador de método **@LogExecution** que:

- Mida cuánto tarda en ejecutarse una función.
- Imprima en consola: `[Log] Ejecutando metodoX - 50ms.`

LECCIÓN 1.8: MÓDULOS Y ORGANIZACIÓN

Arquitectura Limpia y Scalable

En proyectos profesionales, la organización del código es fundamental para la mantenibilidad a largo plazo. **TypeScript** utiliza el sistema de módulos ES6, que permite dividir tu aplicación en archivos pequeños, enfocados y reutilizables.

Patrón Barrel (Barril): Es crucial en proyectos empresariales y monorepos. Consiste en crear archivos index.ts que consolidan las exportaciones de una carpeta. Esto reduce la longitud de los imports y crea una API pública limpia para cada módulo.

```
1 // carpeta: domain/models/index.ts (El "Barrel")
2 export { User } from "./user.model";
3 export { Product } from "./product.model";
4 export { Order } from "./order.model";
5 // Exporta todo de tipos compartidos
6 export * from "./shared.types";
7
8 // --- CONSUMO EN OTRO ARCHIVO ---
9
10 // ✗ MAL: Imports dispersos y rutas largas
11 // import { User } from '../../domain/models/user.model';
12 // import { Product } from '../../domain/models/product.model';
13
14 // ✓ BIEN: Import limpio gracias al Barrel
15 import { User, Product, Order } from "@domain/models";
```

ESTRUCTURA DE PROYECTO PROFESIONAL

Organización por Características (Feature-based): Agrupa archivos relacionados por dominio del negocio (`users/`, `auth/`) en lugar de por tipo técnico (`controllers/`, `services/`). Esto mantiene toda la lógica de una funcionalidad en un solo lugar, facilitando el mantenimiento.

Módulos Desacoplados: Cada módulo debe tener responsabilidades claras. Evita dependencias circulares usando interfaces.

```
1  src/
2    config/          # Variables de entorno y DB
3    common/          # Elementos compartidos globales
4      guards/        # Auth guards
5      filters/       # Manejo de errores
6      pipes/         # Validación
7    modules/         # ORGANIZACIÓN POR DOMINIO
8      auth/          # Módulo de Autenticación
9        strategies/ # JWT Strategy
10       auth.controller.ts
11       auth.service.ts
12      users/         # Módulo de Usuarios
13        dto/          # Data Transfer Objects
14        users.controller.ts
15        users.service.ts
16        users.repository.ts
17      products/       # Módulo de Productos
18    main.ts          # Punto de entrada
```

Esta estructura es el estándar de facto en **NestJS** y arquitecturas escalables.

LECCIÓN 1.9: ASÍNCRONÍA AVANZADA

De Callbacks a Async/Await

El manejo de operaciones asíncronas (Bases de datos, APIs externas) es fundamental. TypeScript proporciona herramientas poderosas para hacerlo sin bloquear el hilo principal.

La Evolución:

1. **Callbacks (Obsoleto)**: Generan "**Callback Hell**". Difíciles de leer y depurar.
2. **Promises (Intermedio)**: Permiten encadenar con `.then()` y manejar errores con `.catch()`.
3. **Async/Await (Moderno)**: Código que parece síncrono pero es asíncrono. Ofrece sintaxis limpia y manejo de errores con `try/catch`. Es el estándar en **TypeScript**.

```
1 // 1. PROMESAS (Verbosidad)
2 function getUser(id: string): Promise<User> {
3     return db
4         .find(id)
5         .then((user) => {
6             return db.getRoles(user.roleId).then((roles) => ({ ...user, roles }));
7         })
8         .catch((err) => console.error(err));
9 }
10
11 // 2. ASYNC/AWAIT (Lectura vertical limpia)
12 async function getUser(id: string): Promise<User> {
13     try {
14         const user = await db.find(id);
15         const roles = await db.getRoles(user.roleId);
16         return { ...user, roles };
17     } catch (err) {
18         console.error(err);
19         throw err;
20     }
21 }
```

PATRÓN AGGREGATOR: PARALELISMO

Orquestación de Servicios

En **microservicios**, a menudo necesitamos construir una vista completa ([ej. "Perfil de Usuario"](#)) obteniendo datos de fuentes dispares (Base de datos de Usuarios + Servicio de Notificaciones).

El Error Común: Hacerlo secuencial ([await user, luego await notifications](#)). Esto duplica el tiempo de espera.

La Solución Profesional: Usar [Promise.all](#) para lanzar las peticiones en paralelo. Si el usuario tarda 200ms y las notificaciones 200ms, en paralelo tardará ~200ms, no 400ms.

```
1 // services/UserProfileManager.ts
2 import { UserService } from "./user.service";
3 import { NotificationService } from "./notification.service";
4
5 export class UserProfileManager {
6   static async getFullUserProfile(userId: string) {
7     try {
8       // 🚀 PARALELISMO: Ambas parten a la vez
9       const [user, notifications] = await Promise.all([
10         UserService.getUserById(userId),
11         NotificationService.getNotificationsForUser(userId),
12       ]);
13
14       // Combinamos los datos solo cuando ambos han llegado
15       return {
16         ...user,
17         notifications: notifications,
18         lastUpdated: new Date(),
19       };
20     } catch (error) {
21       console.error(`Error al armar el perfil: ${error.message}`);
22       throw error; // Re-lanzar para manejarlo arriba
23     }
24   }
25 }
```

OPTIMIZACIÓN: CACHÉ VS API

A veces no queremos esperar a todos, sino al primero que responda.

Escenario: Tenemos un servicio de Caché (rápido pero puede fallar o estar vacío) y una API oficial (lenta pero segura).

Estrategia Promise.race: Lanzamos ambas peticiones. Si la caché responde en 5ms, usamos eso y mostramos el producto inmediatamente, ignorando la API lenta. Es vital para la **UX** en aplicaciones de alto rendimiento.

```
1 // services/ProductAggregator.ts
2
3 export class ProductAggregator {
4     static async getProductFast(id: string): Promise<Product> {
5         console.log("⚡ Iniciando carrera Caché vs API...");
6
7         try {
8             // Retorna el primero que resuelva (o rechace)
9             const result = await Promise.race([
10                 CacheService.get(id), // Típicamente < 50ms
11                 ApiService.getProduct(id), // Típicamente > 200ms
12             ]);
13
14         return result;
15     } catch (error) {
16         // Si el primero falla (ej. caché caída), debemos tener un plan B
17         // Aquí se implementaría la lógica de Fallback
18         throw new Error("Ambos servicios fallaron o timeout");
19     }
20 }
21 }
```

RESILIENCIA CON ALLSETTLED

Promise.all tiene un riesgo: si una sola promesa falla, todo falla. En reportes complejos o dashboards, eso es inaceptable.

Promise.allSettled (ES2020): Ejecuta todo y espera a que terminen, independientemente de si fallaron o no. Te devuelve un array de objetos con el estado (**fulfilled o rejected**).

Caso de Uso: Un panel de control que carga: *Ventas (Éxito), Inventory (Fallo), Usuarios (Éxito)*. Con **allSettled**, mostramos lo que tenemos y un error solo en la parte de Inventory.

```
● ● ●  
1  async function loadDashboard() {  
2      const results = await Promise.allSettled([  
3          SalesService.getDailyStats(),  
4          InventoryService.getStock(),  
5          SupportService.getTickets(),  
6      ]);  
7  
8      results.forEach((result) => {  
9          if (result.status === "fulfilled") {  
10              console.log("✓ Datos cargados:", result.value);  
11          } else {  
12              console.error("✗ Módulo falló:", result.reason);  
13              // Aquí mostraríamos un "placeholder" de error en el UI  
14          }  
15      });  
16  }
```

RESUMEN DE ASÍNCRONÍA

Mejores Prácticas

- **Concurrencia:** Usa `Promise.all` cuando las tareas son independientes. No hagas `await` en bucles `for` (secuencial).
- **Tolerancia a fallos:** Usa `Promise.allSettled` cuando necesites resultados parciales.
- **Rendimiento:** Usa `Promise.race` para establecer Timeouts o competir fuentes de datos.

 **ERROR: EL "CALLBACK HELL" MODERNO** Anidar demasiados `try/catch` o mezclar `.then()` con `await` crea código difícil de leer. Mantén la estructura plana y usa `async/await` siempre que sea posible.

MINI RETO: PROMISE CON TIMEOUT



- 1.Crea una función `promiseWithTimeout<T>(promise: Promise<T>, ms: number)`.
- 2.Debe usar `Promise.race` internamente.
- 3.Compite la promesa original contra una promesa que hace `reject` después de ms milisegundos.
- 4.Si el tiempo se agota, debe lanzar un error "Timeout Error".

SOBRE EL AUTOR

Lewis Oswaldo López Gómez | Arquitecto de Software & FullStack Developer



Soy **Arquitecto de Software** y **Desarrollador FullStack** con más de **10 años de experiencia** construyendo aplicaciones web y sistemas empresariales de **misión crítica**. Durante mi carrera he trabajado en sectores como **banca, educación y SaaS**, siempre con un enfoque muy claro: escribir código mantenable, escalable y preparado para producción.

Mi **stack** principal incluye **TypeScript, Node.js (especialmente NestJS y Express), Angular, React, microservicios en AWS, bases de datos SQL/NoSQL y Docker**. Además, **soy fundador** e ingeniero principal de **MATIAS IMPULSO**, una plataforma SaaS para crear agentes de IA con RAG, y he desarrollado otros productos digitales como **MATIAS ERP, MATIAS API, ASAIE ÉXODO**.

- 🧠 Más de 10 años desarrollando software en entornos reales.
- 🏗️ Especializado en Arquitectura de Software, Microservicios y Cloud (AWS).
- 💻 Experto en **TypeScript, Node.js, NestJS, Angular y React**.
- 🎓 Más de 4 años de experiencia como docente y tutor.
- 🚀 Fundador y creador de varios SaaS en producción.

Mi objetivo con este libro es ayudarte a usar **TypeScript** como una herramienta **profesional**: no solo para “**tipar variables**”, sino para **reducir errores, mejorar la calidad de tu código y diseñar sistemas que puedas mantener y escalar en el tiempo**.

Si quieres seguir aprendiendo conmigo, puedes encontrar más recursos, cursos y contenido avanzado en:

- 🌐 Web: <https://lewislopez.io>
- 🎥 YouTube: [@LEWISLOPEZGOMEZ](https://www.youtube.com/@LEWISLOPEZGOMEZ)
- 📸 Instagram: [@lewislopezgomez](https://www.instagram.com/lewislopezgomez)

¿ESTÁS LISTO PARA EL SIGUIENTE NIVEL?

Has completado los fundamentos de una arquitectura profesional con **TypeScript**. Ahora tienes la base para escribir código que no solo compile, sino **que escale y sobreviva en producción**.

Pero la teoría es solo el **20%**. La verdadera maestría se forja construyendo sistemas reales.

Tu siguiente paso: Dominar Arquitectura Real con NestJS & TypeScript

Lleva lo que aprendiste en este libro a proyectos reales en producción.

Si este Volumen 1 te ayudó a entender **TypeScript** desde una mentalidad más profesional, el siguiente paso lógico es ver cómo todo esto se aplica en arquitecturas reales, con APIs, módulos, dominios, pruebas y despliegues en la nube.

Para eso creé un **Programa Avanzado de NestJS y Arquitectura en Producción**.

¿Qué es este programa?

Un entrenamiento **100% práctico** donde construimos, paso a paso, una **aplicación backend completa con NestJS y TypeScript**, aplicando los principios y patrones que uso en proyectos reales:

- Diseño de módulos y capas (**Domain, Application, Infrastructure**).
- Arquitectura limpia y buenas prácticas en **monolitos y microservicios**.
- Integración con bases de datos (**SQL / NoSQL**).
- Manejo profesional de **errores, logging y seguridad**.
- **Testing** (unitario y de integración) aplicado a NestJS.
- **CI/CD** básico y despliegue en la nube (ej. AWS).

Si este libro te abrió el panorama, el programa avanzado es donde realmente conectamos **teoría con práctica en un entorno profesional**.

[CURSO AVANZADO DE NESTJS Y ARQUITECTURA]

Inscripciones y Mentoría

👉 Haz clic aquí para ver todos los detalles y unirte al programa:

www.lewislopez.io