

# Arquitectura para Frontales con Angular 19+.

<b>Introducción.Qué, por qué y para qué</b>	<b>1</b>
La base del castillo: estructura y carpetas.	2
/core	2
/shared	3
/features	3
Navegando entre componentes sin perder el norte.	3
Lazy load como base de todo.	3
Distribución de ficheros	4
Nomenclatura de ficheros	4
Contenido de un componente (ficheros).	5
Los servicios, protectores del código limpio.	6
Radiografía de un componente.	8
Declaración de variables: Di no a any.	8
Declaración de variables: Ponles nombre como si fueras su padre.	9
Declaración de variables: el orden si afecta al producto.	9
Regiones: un mapa del componente.	10
Comentarios en el código: Di lo justo, no lo evidente.	11
¿Qué, cómo, cuándo y por qué?.	11
CompoDoc, tu compañero de viaje.	12
Comunicación con nuestra API: La unión hace la fuerza.	12
Conectores comunes: centraliza llamadas y evita duplicidades	13
Conectores desde el microscopio. ¿Cómo son?.	13

## Introducción.Qué, por qué y para qué

El objetivo de una arquitectura en el desarrollo de software es unificar y estandarizar la manera en la que todos los programadores codifican, comentan y crean las distintas partes de una aplicación.

Esta estandarización permite agilizar enormemente todas las fases del ciclo de vida de un producto, desde el desarrollo inicial hasta su mantenimiento en producción. Además, incrementa la calidad de los proyectos, reduce el número de errores y facilita la incorporación de nuevos miembros al equipo gracias a una estructura común, documentación clara y guías de trabajo.

Este documento explicará la arquitectura en detalle. Comenzaremos describiendo la estructura general del proyecto, definiendo cómo debe organizarse la distribución de carpetas y ficheros, los diferentes roles dentro de la aplicación y las normas de nomenclatura. Posteriormente, abordaremos la microarquitectura de los componentes, detallando la organización interna, la responsabilidad de cada elemento y las buenas prácticas de codificación.

Una vez sentadas las bases de la estructura, trataremos aspectos esenciales como la seguridad, la gestión de librerías, las políticas de trabajo con Git y otros elementos clave en el desarrollo diario.

Por último, nos centraremos en la gestión de la calidad del código, las pruebas unitarias y E2E, y la integración continua mediante Azure DevOps, asegurando despliegues eficientes, seguros y controlados.

La base de toda arquitectura robusta reside en aplicar los principios de las buenas prácticas de programación, buscando siempre simplificar la estructura y crear componentes lo más atómicos y reutilizables posible.

Cuando hablamos de una aplicación en Angular, la gran mayoría del código, lo vamos a encontrar dentro de la ruta “app/src”.

Aquí encontraremos toda la lógica, páginas, servicios y otros elementos tanto comunes como privados que darán forma a nuestra app.

## La base del castillo: estructura y carpetas.

Desde este primer nivel ya podemos comenzar a dar forma a nuestra arquitectura, intentando separar de manera lógica cada funcionalidad. Aquí podemos ver una representación general de cómo quedaría la base de nuestro proyecto.

```
/src
├── /app
│   ├── /core           # Servicios, guardias e interceptores globales
│   ├── /shared         # Componentes, directivas y pipes reutilizables
│   ├── /features       # Módulos principales de funcionalidad
│   ├── app-routing.module.ts
│   └── app.config.ts
```

### /core

Nos permite almacenar diferentes servicios comunes como interceptores, guardias de rutas, configuraciones globales... Aquí hay que destacar que sólo se almacenarán elementos comunes a toda la aplicación, evitando al máximo crear lógica específica para un componente. También hay que destacar que en los ficheros que almacenemos aquí, es especialmente importante la limpieza de código y el orden, ya que al poder realizar trabajos muy genéricos y dispares, su mantenimiento puede llegar a volverse complicado.

## /shared

Aquí podremos almacenar todos los elementos “visuales” comunes a la aplicación. Generalmente se utiliza para almacenar pipes comunes, directivas y componentes que van a utilizarse en diferentes sitios de nuestra aplicación. Más adelante veremos que debe contener un componente y cómo debe comportarse, pero como aperitivo, dejar muy claro que estos componentes deben ser **comunes**, es decir, no deben contener ningún tipo de lógica propia más allá de la necesaria para ejecutar su función (por ejemplo, si es un formulario cuyos datos se envían a una API, los procesos de transformación no se realizan aquí).

## /features

Será nuestra carpeta principal, en la que más tiempo pasamos y la encargada de almacenar el grueso de nuestro código. Aquí encontraremos las diferentes pantallas de nuestra aplicación, distribuida por funcionalidades y agrupando de manera lógica los componentes.

Como podemos observar, **shared** y **core** comparten ciertas similitudes, ya que ambos agrupan elementos comunes reutilizables en toda la aplicación. Sin embargo, en caso de duda sobre dónde añadir un nuevo servicio o componente, podemos establecer una regla general:

**core** está orientado principalmente a servicios, utilidades y elementos no visuales, enfocados en la seguridad, la infraestructura y el funcionamiento interno de la aplicación. Por otro lado, **shared** se centra en elementos visuales y componentes de interacción con el usuario, como directivas, pipes y componentes genéricos reutilizables en distintas partes de la interfaz.

## Navegando entre componentes sin perder el norte.

A partir de este apartado definiremos las bases de **estructura**, **nomenclatura** y **organización** que deben aplicarse de forma general en toda la aplicación.

Aunque estas normas son aplicables a todas las carpetas que hemos visto anteriormente, será en la carpeta **features** donde este orden y consistencia cobren especial importancia, ya que es donde se concentrará la mayor parte del código de negocio y las pantallas de la aplicación.

Mantener una estructura limpia y coherente en **features** será clave para garantizar un desarrollo ágil, facilitar el mantenimiento y asegurar la escalabilidad del proyecto.

## Lazy load como base de todo.

Cuando hablamos de la distribución de los componentes de una aplicación en Angular, siempre debemos partir de la base de **Lazy Load**. Pero, ¿qué significa Lazy Load? En términos sencillos, hace referencia a la estrategia de carga **diferida** de los módulos dentro de la aplicación.

Este enfoque es crucial, ya que una mala estrategia de routing o la carga excesiva del módulo principal (**app.module.ts**) o de módulos hijos puede hacer que Angular cargue demasiados elementos al acceder a ciertas secciones, lo que retrasaría el acceso a esas partes de la app y empeoraría la experiencia del usuario.

**Lazy Load** permite cargar **sólo los módulos necesarios en el momento exacto**, distribuyendo la carga total de la aplicación en pequeños bloques, lo que mejora la velocidad de carga y la **fluidez** de la aplicación. Esto no solo hace que la app se sienta más rápida, sino que optimiza el uso de recursos, ya que solo se carga lo que se necesita en cada momento.

Aunque la organización de los ficheros en sí misma no afecta directamente a la estrategia de Lazy Load, la correcta **distribución y agrupación de componentes** siguiendo esta lógica hará que la aplicación sea mucho más fácil de comprender y mantener. Así, podremos encontrar el componente que necesitamos de manera más rápida y con mayor claridad, mejorando la estructura general del proyecto.

Hasta ahora hemos hablado a grandes rasgos de cómo debería distribuirse la aplicación, pero a partir de aquí dejamos atrás las generalidades y entramos de lleno en la distribución real de los componentes.

## Distribución de ficheros

Lo primero que debemos recalcar es la necesidad de que nuestra aplicación esté diseñada para funcionar desde el primer momento bajo la filosofía de Lazy Load, y que la distribución de sus ficheros acompañe esta idea.

La agrupación principal será compuesta por módulos funcionales. Estos módulos serán fácilmente identificables, ya que estarán centrados en realizar una única función de negocio. Por ejemplo, podemos tener un módulo **Cientes**, dentro del cual se gestiona el listado, alta, edición y borrado de clientes. Todas estas funcionalidades comparten un mismo objetivo: gestionar clientes en sus diferentes estados, por lo que constituirán nuestra primera división lógica.

## Nomenclatura de ficheros

Es habitual que a la hora de nombrar los ficheros surjan dudas sobre hasta qué punto ser específicos o repetitivos. Por ejemplo, dentro del módulo **Cientes**, podríamos llamar al listado simplemente list y al formulario add-edit. Debemos evitar que el nombre del fichero (que no el del componente) arrastre toda la raíz del módulo (por ejemplo, clientes-gestion-alta.component.ts), ya que cuanto más profundo sea el árbol, más largos e incómodos se volverán los nombres de los archivos.

En cambio, a nivel de **nombres de componentes**, sí podemos ser más específicos y descriptivos, de forma que a la hora de importarlos, su propósito quede completamente claro.

A continuación, se muestra un ejemplo básico de la estructura de un módulo **Cientes**, el cual a su vez puede tener hijos que siguen esta misma organización. No existe un límite

estricto de niveles de anidación, aunque tener más de tres niveles suele ser algo excepcional reservado a zonas especialmente complejas.

```
/clientes
├── clientes.module.ts      # Módulo principal de la funcionalidad 'Clientes'
├── clientes-routing.module.ts # Configuración de rutas de 'Clientes'
├── clientes.component.ts   # Componente raíz de 'Clientes'
├── clientes.component.html  # Template principal
├── clientes.component.scss  # Estilos del componente
├── clientes.component.spec.ts # Test unitario
├── /alta-edicion b
│   ├── alta-edicion.component.ts
│   ├── alta-edicion.component.html
│   ├── alta-edicion.component.scss
│   └── alta-edicion.component.spec.ts
├── /listado
└── /delete
```

## Contenido de un componente (ficheros).

Poco a poco vamos entrando en las zonas más profundas de la aplicación, y por lo tanto, cada vez es más importante ser meticulosos a la hora de generar nuestro código.

Cuando generamos un componente en Angular, se nos crean automáticamente los archivos básicos (.html, .scss o .sass, y .ts), pero esto no significa que nuestro trabajo haya terminado. Todo lo contrario: ahora es cuando empieza la verdadera planificación.

Debemos pensar cómo será nuestro componente:

- ¿Qué apartados tendrá?
- ¿Usará componentes comunes o todo será interno?
- ¿Qué dependencias necesitará?

Una de las primeras decisiones importantes es: **¿Nuestro componente necesita un módulo propio?**

Para responder a esto debemos preguntarnos: *¿Tendremos varias vistas relacionadas que deban ser agrupadas bajo el nombre de "Clientes"?*

Si la respuesta es sí, entonces nuestro componente debería tener su propio módulo, sus rutas internas, y mantener independencia respecto al resto de la aplicación.

Otro punto fundamental es decidir **cuándo utilizar componentes Standalone y cuándo no**.

Por norma general, **Standalone** debería utilizarse en dos situaciones principales:

- Cuando creamos un **componente común** que se utilizará en distintas pantallas de un mismo módulo, pero no fuera de él.
- Cuando el componente es **especialmente grande o dependiente de librerías pesadas** que no se usan en otros lugares. En este caso, crearlo como Standalone ayuda a que la aplicación solo cargue esas dependencias cuando realmente se necesitan, mejorando así el rendimiento general.

Como resumen, muestro esta tabla rápida que permite ver de un vistazo en que situaciones usaremos un componente Standalone y en cual un módulo.

Caso	Crear Módulo	Crear Componente Standalone
Varias pantallas relacionadas	Si	No
Necesidad de rutas internas	Si	No
Un solo componente aislado	No	Si
Componente usado solo dentro de un módulo	No	Si
Componente muy pesado (muchas librerías)	No	Si
Componente pequeño y específico	No	Si
Componente que debe ser fácilmente reutilizable	No	Si
Necesidad de lazy load para varias vistas	Si	No

### Los servicios, protectores del código limpio.

Cuando hablamos de buenas prácticas, una de las más importantes es “atomizar” el componente, o dicho de otra manera, que un componente tenga una funcionalidad muy limitada, centrándose única y exclusivamente en su trabajo (un componente de formulario pinta y almacena sus datos, válida errores y se encarga de que visualmente todo este correcto, pero no se adapta a un modelo de la API, tiene la funcionalidad de guardar elementos o gestionar otras funciones). Sin embargo, por muy pequeños que hagamos nuestros componentes, tendremos un montón de código que “ensucia” nuestro código limpio.

Un componente debe encargarse de manipular las vistas, almacenar la información mínima necesaria para funcionar y emitir datos a padres e hijos.

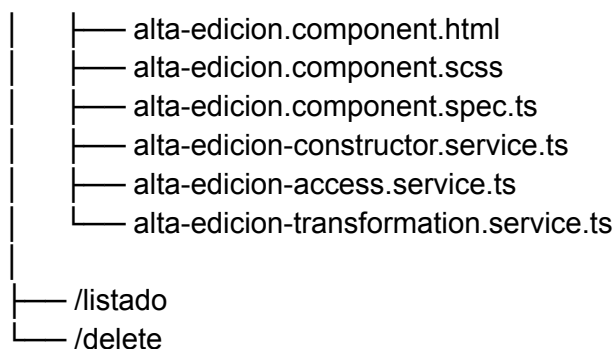
¿Qué ocurre si necesitamos transformar un objeto, realizar validaciones complejas o hacer llamadas a APIs? Para ello, esta arquitectura propone un estándar de tres servicios por componente (ampliables si fuese necesario, aunque si superamos ampliamente estos tres suele ser una señal de que nuestro componente no es suficientemente atómico).

Estos servicios son:

- Constructor: este servicio nos permitirá inicializar todas nuestras librerías, objetos... Gracias a esto, tendremos agrupadas toda la lógica de inicialización, permitiendo ahorrarnos muchas líneas en nuestro componente principal y haciendo que cuando este crezca mucho, no tener que andar navegando entre líneas dedicadas a inicializar una tabla, la cual lleva sin tocarse mucho tiempo.
- Access: su función principal es el acceso a datos a través de una API (por si hay alguna duda, Angular no accede a la BBDD directamente bajo ninguna circunstancia). Estandariza todas las llamadas, haciendo que puedas devolver objetos adaptados a tus necesidades y gestionar los errores.
- Transformation: seguramente sea el más utilizado e importante. Su función principal es hacer todo tipo de cálculos, validaciones, transformaciones o en general, cualquier lógica que no requiera trabajar con elementos visuales del componente.

Esta segmentación del código marcará la diferencia cuando hablamos de organización, haciéndonos mucho más fácil añadir o modificar código en nuestro componente, ya que sabemos exactamente donde tenemos el acceso a los datos, su transformación o su inicialización. Al principio esto puede parecer engorroso, más lento o generar ficheros de forma innecesaria, pero recordar que en la época en la que estamos, aligerar una web en 200 líneas, son unos kb totalmente imperceptibles para el usuario, sin embargo, el tiempo que nos ahorra desarrollando es enorme. Así quedaría nuestro famoso componente de clientes con los añadidos de los servicios.

```
/clientes
|
|— clientes.module.ts
|— clientes-routing.module.ts
|— clientes.component.ts
|— clientes.component.html
|— clientes.component.scss
|— clientes.component.spec.ts
|
|— clientes-constructor.service.ts
|— clientes-access.service.ts
|— clientes-transformation.service.ts
|
|— /alta-edicion
|   |— alta-edicion.component.ts
```



Esta estructura facilita el desarrollo y el mantenimiento de nuestro código enormemente, facilitando la navegación en los componentes y sus interacciones entre ellos.

## Radiografía de un componente.

Con lo visto hasta ahora, ya podemos presumir de un proyecto al cual, no asusta entrar. La búsqueda de componentes es sencilla, ubicar un error es sencillo puesto que sabes si es un error de construcción de datos, de la llamada a la api o de transformación de datos y tu aplicación va fluida gracias a la carga distribuida con tus módulos y componentes standalone. Sin embargo, seguimos teniendo un monstruo debajo de la cama, el temible spaghetti code, causante de dolores de cabeza, subidas a producción que parecen arreglar una cosa y romper 2 y otros efectos secundarios de un código perezoso.

Es fundamental que un componente esté correctamente definido, con una planificación global entre todos los equipos para que la forma en la que se programa, el orden, la nomenclatura de variables... permite una mejor cohesión entre todos, por lo que vamos a establecer una serie de pautas que nos ayudarán a estandarizar la forma en la que programamos. Más adelante también veremos cómo ESLint puede ayudarnos a mantener el código en buena forma.

## Declaración de variables: Di no a any.

En este apartado nos centraremos en cómo, dónde y por qué debemos declarar las distintas variables que maneja nuestro componente.

La norma más básica e importante es sin duda alguna el tipado de las variables. No existe ninguna buena razón para no tipar las variables. Inicialmente puede parecer más rápido declarar un montón de variables como `any`, `[]` o `{}`. Sin embargo, esto dificultará enormemente el trabajo en componentes complejos. También supondrá un problema muy grande las funciones que retornan datos no tipados, por lo que siempre debemos asegurarnos de que las variables tienen un tipo asociado.

Cuando trabajemos con librerías ajenas a nosotros, las cuales tengan **EventEmitter** sin tipar, nosotros debemos crearnos una carpeta con los distintos modelos que vamos a utilizar en el componente, o bien creando una carpeta **models** dentro de nuestro componente, o utilizando una librería común a todos nuestros proyectos, facilitando el trabajo a nuestros compañeros.



## Declaración de variables: Ponles nombre como si fueras su padre.

En los componentes, a menudo necesitamos muchas variables, y todos hemos tenido que desplazarnos por el código para encontrar la declaración de la que necesitamos trabajar. Esto puede hacer que el desarrollo sea más lento, nos haga perder el foco y dificulta la depuración. Para facilitar estas tareas, estandarizar los nombres de variables y funciones con un patrón lógico nos ayudará a trabajar de manera más eficiente.

Una nomenclatura coherente y clara facilita la legibilidad del código y hace que tanto los desarrolladores actuales como los futuros puedan comprender rápidamente el propósito de cada variable y su uso dentro del componente

- Comienzo indicando que es esa variable: las variables deberán indicar en su inicio que son, es decir, `input`, `select`, `form`, `model`... no es necesario especificar su tipo, ya que al estar tipadas, esa información la tendremos.
- ¿Qué función realiza?: para la parte central, indicaremos cual es la función de esa variable, por ejemplo, una encargada de almacenar el valor de un campo nombre será `inputName`—.
- Como se va a utilizar: para terminar la variable, debemos usar sufijos **CR** (para referencias de `ViewChild`), **data** (para las variables que almacenan información) y **control** (para elementos que controlan visualización permisos...). Los demás tipos de variables no necesitan llevar un sufijo. Por ejemplo tendríamos ejemplos como `@ViewChild('inputNameVC') private inputNameCR: ElementRef;`  
`private inputNameData: string;`  
`private formLoginControl: FormControl;`
- Camel case siempre: es el formato más aceptado por todos. Todos los nombres siempre en inglés.

Estas normas nos permiten dar nombres a las variables mucho más claras, lo que nos llevará a programar más eficiente. También aporta cohesión al trabajo de todos los equipos, haciendo mucho más sencillo revisar el código de otra persona o tener que entender su componente.

## Declaración de variables: el orden si afecta al producto.

Para cerrar el capítulo de la declaración de variables, es momento de hablar de un aspecto crucial que, aunque no afecta a la ejecución del código, sí tiene un impacto directo en la experiencia de desarrollo: **el orden en el que las declaramos**.

Un orden lógico y estandarizado facilita enormemente la lectura del código, acelera el desarrollo y reduce errores. No es lo mismo perder cinco minutos buscando una variable en medio del caos que encontrarla en el primer scroll. Si tus compañeros pueden intuir dónde está algo antes de buscarlo, estás haciendo bien las cosas.

Este orden propuesto agrupa las variables por su finalidad en la pantalla. No sustituye al uso de regiones (que recomendamos en componentes grandes), sino que lo complementa. Vamos a verlo:

- **@Input/output:** Primero lo que entra y lo que sale. Es decir, las propiedades públicas que exponen la API del componente. Así se sabe rápidamente qué necesita y qué emite.
- A continuación, todas las referencias al HTML, ya sea mediante decoradores de Angular o búsquedas directas (`getElementById`, `querySelector`, etc.). Agrupar esto permite ver fácilmente qué partes del DOM estamos manipulando.
- Aquí agrupamos todos los flags, booleanos o variables que controlan la visibilidad o estado visual de partes del componente: `isLoading`, `showModal`, `hasPermission`, etc.
- Variables que contienen la información con la que trabaja el componente: datos de API, formularios, listas, modelos intermedios... En resumen, todo lo que guarda o procesa datos reales.
- **Miscelánea:** Por último, cualquier variable que no encaje claramente en los bloques anteriores: contadores, identificadores, settings internos, helpers temporales...

Aunque parezca un detalle menor, **el orden de las variables marca la diferencia entre un componente mantenible y una trampa mortal**. Sigue este patrón y combínalo con nomenclaturas claras y regiones, y tus compañeros te querrán (o al menos no te odiarán).

## Regiones: un mapa del componente.

Por muy pequeño, preciso y optimizado que queramos hacer nuestro componente, tenemos que asumir que ciertas partes de la aplicación son muy complejas, por lo el tamaño de ciertos componentes crecerá mucho, haciéndolos más difíciles de trabajar con ellos. En estas situaciones las **regiones** vienen a salvarnos la vida.

Las regiones son agrupaciones lógicas de código sin funcionalidad en la ejecución del código, pero que nos permiten organizar nuestro componente para facilitar la lectura del código. Esta arquitectura propone 4 bloques principales en el componente:

- **Inicialización de datos:** se encarga de inicializar nuestras variables, formularios, librerías y cualquier elemento de nuestro componente que necesite configurarse. Siempre llamarán a métodos de inicialización de nuestro servicio de construcción.
- **Funciones de callback:** reciben los eventos de emisión de los componentes hijos, permitiéndonos obtener y asignar datos, emitir eventos o modificar la vista.
- **Funciones comunes:** aquí estarán todas las funciones que se encargan de conectar los distintos flujos de nuestro componente. Estas funciones son privadas ya que no se les llamara desde el html.
- **Acceso a datos:** se encargará de llamar a nuestro servicio de acceso a datos para obtener toda la información de nuestra API.

Con estos 4 módulos nuestro componente será como un libro abierto, pudiendo saber de forma rápida donde se encuentra el método al que queremos acceder. Aunque no es obligatorio, en componentes complejos, es muy recomendable que la declaración de variables también esté agrupada en regiones. En este caso no hay una estructura tan

definida, ya que cada componente es distinto, pero es muy recomendable anidar estas declaraciones siguiendo un esquema como este:

- variables de control visual de html: permiten mostrar y ocultar diferentes elementos.
- variables de almacenamiento de datos: datos de formularios, ficheros, variables de control...

## Comentarios en el código: Di lo justo, no lo evidente.

Con lo visto hasta ahora, podemos presumir de un código limpio, eficiente y de lectura muy fácil, sin embargo, tenemos que enfrentarnos a una realidad incómoda, **NO TODOS LOS MÉTODOS VAN A SER COSER Y CANTAR**. A lo largo del desarrollo de nuestro producto nos encontraremos funciones con lógicas complejas, largas (muchas líneas) y métodos que se encadenan unos con otros, recibiendo diferentes entradas y múltiples salidas. Para esos métodos (y sus variables) tenemos el arma definitiva: La documentación.

### ¿Qué, cómo, cuándo y por qué?.

Siempre que declaremos una variable o un método debemos plantearnos estas preguntas. Esto nos servirá para no llenar el código de comentarios, ya que en este caso, la falta de comentarios es tan odiosa como el exceso.

Un código comentado con cabeza nos facilita volver a trabajar con el componente meses después de haberlo programado, facilita la adaptación de nuevos miembros al equipo y agiliza enormemente el desarrollo al saber exactamente que ciclos de vida toca el método que estamos tocando.

- **¿Qué?:** Todo código es susceptible a tener un comentario asociado, da igual que sean variables, métodos, bloques HTML, código CSS... Los comentarios son versátiles y nos ayudan a recordar lógicas enrevesadas y casos de uso complejos, por lo que no te cortes a la hora de comentar cualquier parte de tu código, si crees que aporta, adelante.
- **¿Cómo?:** Usando la cabeza, un comentario debe aportar valor, debe sumar información útil y relevante sobre el código en cuestión. Un método que inicializa una variable a un **string** vacío no necesita explicación ya que es una sola línea, sin lógica compleja y dentro de un método cuya nomenclatura es explícita (`private initInputNameData()`).
- **¿Cuándo?:** para responder esta pregunta es bueno hacerte a ti mismo otras. ¿Te ha costado programar esa funcionalidad?, ¿Mientras lo hacías has tenido problemas con los parámetros de entrada y salida?, ¿Si ese método falla se rompe toda la ejecución del componente?. Si la respuesta a cualquiera de estas preguntas es Sí, entonces ya tienes tu respuesta, ese método debe tener una documentación breve y concisa que explique a otros programadores (o a tu yo del futuro) como se debería comportar.
- **¿Por qué?:** Esta pregunta variará enormemente entre cada método. Tu objetivo aquí es responder a por qué funciona así tu método. Qué valores esperas en los input de la función, por qué retornas ese valor, por que se asignan x valores a x variables o

por que tienes ese If. Muy resumidamente se podría decir que es la explicación de tu método.

## CompoDoc, tu compañero de viaje.

Si hablamos de un código bien comentado, no podemos dejar pasar la oportunidad de mejorar la documentación de nuestra aplicación de forma indirecta. Cualquier persona que haya trabajado en proyectos grandes se habrá enfrentado a documentaciones como Métrica V3, PRINCE2... Documentos necesarios, sí, pero tediosos de generar y, sobre todo, de mantener.

Compodoc no pretende sustituir estas metodologías, pero sí nos permite generar de forma rápida y dinámica una documentación técnica del código. Ofrece una visión general del proyecto, su estructura, entradas/salidas, rutas, relaciones entre módulos, servicios, componentes... todo basado en los comentarios y anotaciones que ya escribimos al desarrollar.

Si mantenemos actualizados los bloques de comentarios, con un solo comando podremos generar una página web con la documentación viva del proyecto.

Compodoc se instala fácilmente desde NPM, se integra sin complicaciones en proyectos Angular y es muy rápida de aprender. En otros documentos entraremos más a fondo en su uso y configuración, pero quería aprovechar este punto para recomendar que le echéis un vistazo. Puedes consultar más información en su [sitio web oficial](#).

Como resumen final de este bloque, conviene recordar que **los comentarios también pueden ser contraproducentes si se abusa de ellos**. Además, debemos tener cuidado al usar herramientas de IA como Copilot, ChatGPT u otras. Son excelentes para ayudarte a redactar mejor un comentario ya existente o revisar si aporta suficiente contexto, pero **no es recomendable pedirles directamente que comenten tu código**. Suelen generar explicaciones innecesarias, genéricas o redundantes, inflando el archivo y reduciendo el valor real de los comentarios. Lo peor de todo: acostumbran a que el siguiente desarrollador ignore los comentarios porque "nunca dicen nada útil".

## Comunicación con nuestra API: La unión hace la fuerza.

Cuando hablamos de frontales, no podemos olvidarnos de que nosotros somos el 50%, ya que el otro 50% está compuesto por nuestros compañeros del Back, con sus APIS, BBDD y otros elementos sin los cuales, nuestra página valdría para poco.

Todas las páginas web se apoyan en una API a la que atacar para obtener, guardar y gestionar los datos de nuestros clientes, por lo que cuando hablamos de arquitectura de frontales, no podemos pasar por alto cómo organizar, gestionar y mantener todas esas llamadas es necesario tener un plan sólido, que ceda las responsabilidades a quien toca en cada momento.

## Conectores comunes: centraliza llamadas y evita duplicidades

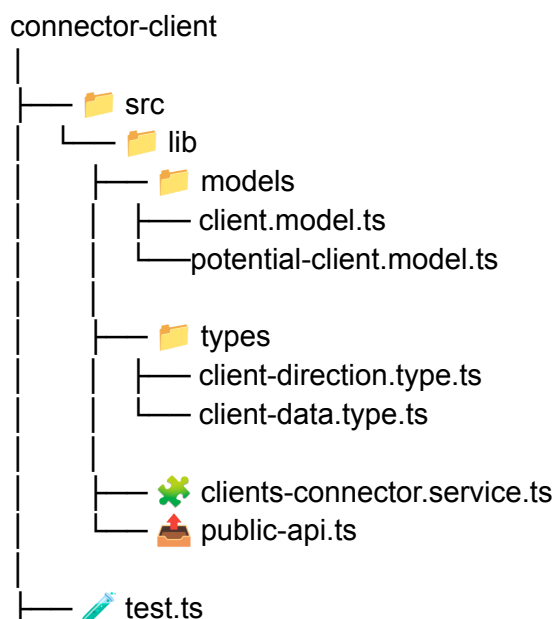
En nuestra aplicación, algunas llamadas a la API serán específicas de una sola pantalla, pero muchas otras se reutilizarán en distintas partes del proyecto, o incluso en otros proyectos. Por eso, lo más adecuado es crear pequeñas librerías propias que agrupen esas llamadas, junto con sus modelos y cualquier lógica auxiliar necesaria. Así, si el backend cambia un modelo, endpoint o cabeceras, solo habrá que modificar la librería una vez. Al publicarla y actualizarla en los proyectos que la usen, el impacto será mínimo.

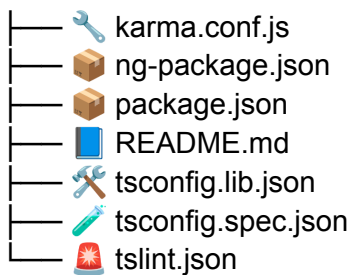
### Conectores desde el microscopio. ¿Cómo son?

Las funciones que desarrollamos en estos conectores deben ser lo más genéricas posible y no contener lógica de negocio. Su responsabilidad debe limitarse a construir la petición HTTP con los parámetros necesarios. La única excepción serían métodos auxiliares, como la construcción de QueryStrings, que se pueden ubicar en una librería compartida entre conectores, para que todos utilicen el mismo formato.

Las funciones que programaremos aquí deben ser genéricas y sin lógica, es decir, solo recibirán parámetros y los añadirán a la llamada, nada de transformar, calcular u otras cuestiones. La **ÚNICA** excepción serán métodos auxiliares como puede ser la creación de QueryString para adaptarse al estándar Http rest,. Todas estas librerías podrán tener una librería común a ellas con este tipo de métodos de transformación para evitar que cada programador lo haga a su manera. Se deberá usar un modelo de construcción de queryString común a todos.

En estas librerías también tendremos una carpeta de modelos (models) en las cuales guardaremos todos los modelos tanto para enviar al servidor como para recibirlos. Esto facilitará enormemente el trabajo de tipar nuestras variables en el proyecto principal, ya que como todos estos modelos estarán exportados desde el public-api.ts, con hacerlos una vez lo tendremos disponible en todos nuestros proyectos. La imagen de un conector será algo similar a esto:





Como se puede ver, estas librerías son ligeras, fáciles de mantener e integrar. Suponen una gran diferencia entre trabajar en silos o tener un ecosistema de equipos que reutilizan código, comparten responsabilidades y estandarizan sus integraciones con la API.

## Módulo de Transporte HTTP Genérico.

Tras este nombre tan aparatoso se esconde una pieza clave de nuestra arquitectura: **la librería común de transporte REST**. Esta librería centraliza los principales métodos de comunicación HTTP que necesita nuestra aplicación, permitiendo **agrupar y estandarizar** todas las llamadas a API según las convenciones de la empresa.

Una de sus funciones más importantes es **adaptar las respuestas del backend** a un formato uniforme para el frontend. Esto significa que, aunque distintas APIs devuelvan objetos con estructuras diferentes, gracias a esta capa de transformación, el frontend siempre trabajará con un modelo de datos común, facilitando la reutilización y el mantenimiento.

Además, esta librería permite **gestionar cabeceras comunes** (como tokens, idiomas, versiones, etc.) de forma centralizada. En lugar de añadirlas manualmente en cada petición, se configuran una vez y se aplican automáticamente a todas las llamadas.

Otro de sus puntos fuertes es que puede incluir **interceptores personalizados**, como por ejemplo:

- Inyección automática del token de seguridad.
- Redirección automática en errores 401 o 403.
- Captura y gestión de errores del backend.
- Logs o métricas de rendimiento de las llamadas.

Gracias a este enfoque, conseguimos una **arquitectura más limpia, mantenible y robusta**, donde el frontend se desacopla de las particularidades de cada API y trabaja con una capa de transporte coherente y confiable.