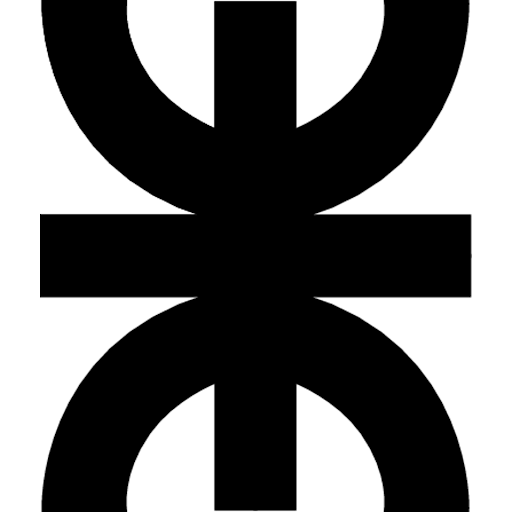
Universidad Tecnológica Nacional

# Facultad Regional Córdoba

Ingeniería en Sistemas de Información

****

# Cátedra: Ingeniería de Software

## Trabajo Práctico N°6

### Documento de buenas prácticas

## INTEGRANTES

94.093 NIELSEN PELLINACCI, J. Manuel

82.043 FARACE, Florencia Candelaria

80.122 RICSE Rojas, Javier Alejandro

82.186 NOBILE, Valentina

82.376 CARGNELUTTI, Clever Lautaro

82.742 BARRAGÁN, Abril

# Documento de Estilo de Código.

Para trabajar con **Angular CLI**, seguimos una serie de buenas prácticas detalladas en su documentación.

## Regla de Uno

Defina una cosa, como un servicio o componente, por archivo. Considere limitar los archivos a 400 líneas de código.

¿Por qué? Un componente por archivo:

* Hace que sea mucho más fácil de leer, mantener y evitar colisiones con los equipos de control de código fuente.
* Evita errores ocultos que a menudo surgen al combinar componentes en un archivo donde pueden compartir variables, crear cierres no deseados o acoplamiento no deseado con dependencias.
* Hace que un solo componente puede ser la exportación predeterminada para su archivo, lo que facilita la carga diferida con el enrutador.

La clave es hacer que el código sea más reutilizable, más fácil de leer y menos propenso a errores. Es una mejor práctica redistribuir el componente y sus clases de soporte en sus propios archivos dedicados.

## Pequeñas Funciones

Defina funciones pequeñas. Considere limitar a no más de 75 líneas.

**¿Por qué?** Las funciones pequeñas:

* Son más fáciles de probar, especialmente cuando hacen una cosa y tienen un propósito.
* Promueven la reutilización.
* Son más fáciles de leer y mantener.
* Ayudan a evitar errores ocultos que vienen con funciones grandes que comparten variables con alcance externo, crean cierres no deseados o acoplamientos no deseados con dependencias.

## Directrices generales de nomenclatura

Utilice nombres coherentes para todos los símbolos. Siga un patrón que describa la característica del símbolo y luego su tipo. El patrón recomendado es *feature.type.ts.*

**¿Por qué?**

* Las convenciones de nomenclatura ayudan a proporcionar una forma consistente de encontrar contenido de un vistazo. La consistencia dentro del proyecto es vital. La consistencia con un equipo es importante. La coherencia en toda la empresa proporciona una enorme eficiencia.
* Las convenciones de nomenclatura deberían simplemente ayudar a encontrar el código deseado más rápido y hacerlo más fácil de entender.
* Los nombres de carpetas y archivos deben transmitir claramente su intención. Por ejemplo, *app/heroes/hero-list.component.ts* puede contener un componente que gestione una lista de héroes.

## Símbolos y nombres de archivos

Utilice nombres coherentes para todos los activos que lleven el nombre de lo que representan.

Utilice mayúsculas y minúsculas en camello para los nombres de las clases.

Haga coincidir el nombre del símbolo con el nombre del archivo.

Agregue el nombre del símbolo con el sufijo convencional (como *Component*, *Directive*, *Module*, *Pipeo Service*) para una cosa de ese tipo.

Asigne al nombre de archivo el sufijo convencional (como *.component.ts, .directive.ts, .module.ts, .pipe.tso .service.ts*) para un archivo de ese tipo.

¿Por qué? Las convenciones coherentes facilitan la identificación y referencia rápidas de activos de diferentes tipos.

## Arranque

Coloque el arranque y la lógica de la plataforma para la aplicación en un archivo llamado *main.ts*. Incluya el manejo de errores en la lógica de arranque. Evite poner la lógica de la aplicación en *main.ts*. En su lugar, considere colocarlo en un componente o servicio.

¿Por qué?

* Sigue una convención coherente para la lógica de inicio de una aplicación.
* Sigue una convención familiar de otras plataformas tecnológicas.

## Estructura LIFT

Estructure la aplicación de modo que pueda ***L***ocalizar el código rápidamente, ***I***dentificar el código de un vistazo, ***M***antener la estructura más plana que pueda e ***I***ntentando estar seco\*. Defina la estructura para seguir estas cuatro pautas básicas, enumeradas en orden de importancia.

**¿Por qué?** LIFT proporciona una estructura consistente que escala bien, es modular y facilita el aumento de la eficiencia del desarrollador al encontrar el código rápidamente. Para confirmar su intuición sobre una estructura en particular, pregunte: ¿puedo abrir rápidamente y comenzar a trabajar en todos los archivos relacionados para esta característica?

\* Estas siglas se corresponden con las siglas de *Localize*, *Identify*, *Flat* y *Try* en inglés.

### Localizar (Locate)

Haga que la localización del código sea intuitiva, simple y rápida.

**¿Por qué?** Para trabajar de manera eficiente, debe poder encontrar archivos con rapidez, especialmente cuando no sabe (o no recuerda) los nombres de los archivos. Mantener los archivos relacionados cerca unos de otros en una ubicación intuitiva ahorran tiempo. Una estructura de carpetas descriptiva marca una gran diferencia para usted y las personas que vienen después de usted.

### Identificar (Identify)

Nombra el archivo de manera que sepas al instante lo que contiene y representa. Sea descriptivo con los nombres de archivo y mantenga el contenido del archivo en exactamente un componente. Evite archivos con múltiples componentes, múltiples servicios o una mezcla.

**¿Por qué?** Pase menos tiempo buscando y picoteando el código, y sea más eficiente. Los nombres de archivo más largos son mucho mejores que los nombres abreviados cortos pero oscuros.

### Plano (Flat)

Mantenga una estructura de carpetas plana el mayor tiempo posible.

Considere la posibilidad de crear subcarpetas cuando una carpeta llegue a siete o más archivos. Además, considere la posibilidad de configurar el IDE para ocultar archivos irrelevantes que distraigan, como archivos generados *.jsy .js.map*.

**¿Por qué?**

* Nadie quiere buscar un archivo a través de siete niveles de carpetas.
* Una estructura plana es fácil de escanear.
* Por otro lado, los psicólogos creen que los humanos comienzan a tener problemas cuando el número de cosas interesantes adyacentes supera las nueve. Entonces, cuando una carpeta tiene diez o más archivos, puede ser el momento de crear subcarpetas.

Base su decisión en su nivel de comodidad. Use una estructura más plana hasta que haya un valor obvio para crear una nueva carpeta.

### Tratar de estar seco (Try to be Dry)

Sea SECO (No se repita). Evite estar tan SECO que sacrifique la legibilidad.

**¿Por qué?** Estar SECO es importante, pero no crucial si sacrifica los otros elementos de LIFT (por eso es *tratar* de estar seco). Por ejemplo, es redundante nombrar una plantilla *hero-view.component.html* porque con la extensión *html*, obviamente es una vista. Pero si algo no es obvio o se aparta de una convención, entonces explíquelo.

## Estructura de carpetas por características

Cree carpetas con el nombre del área característica que representan.

**¿Por qué?**

* Un desarrollador puede localizar el código e identificar lo que representa cada archivo de un vistazo. La estructura es lo más plana posible y no hay nombres repetitivos o redundantes.
* Todas las pautas de LIFT están cubiertas.
* Ayuda a evitar que la aplicación se abarrote organizando el contenido y manteniéndolo alineado con las pautas de LIFT.
* Cuando hay muchos archivos, por ejemplo más de 10, localizarlos es más fácil con una estructura de carpetas coherente y más difícil con una estructura plana.

Cree un *NgModule* para cada área de funciones ya que *NgModules* facilita la carga diferida de funciones enrutables, el aislamiento, la prueba y la reutilización de funciones.

## Secuencia de miembros

Coloque las propiedades en la parte superior seguidas de los métodos.

Coloque los miembros privados después de los miembros públicos, en orden alfabético.

**¿Por qué?** La colocación de los miembros en una secuencia coherente facilita la lectura y ayuda a identificar instantáneamente qué miembros del componente sirven para qué propósito.