

Reglas de estilo de programación en Angular

Código del documento

24/02/2020

Dawid Mateusz

Departamento IT

Contenido

[1. Introducción 4](#_Toc65052318)

[2. Reglas de arquitectura 4](#_Toc65052319)

[2.1 Organización de directorios 4](#_Toc65052320)

[3. Componentes 5](#_Toc65052321)

[3.1 Nomenclatura de carpetas 5](#_Toc65052322)

[3.2 Estructura de carpetas 5](#_Toc65052323)

[3.3 Nomenclatura de archivos 6](#_Toc65052324)

[3.4 Estructura de archivos 6](#_Toc65052325)

[3.5 Nomenclatura de clase 6](#_Toc65052326)

[4. Reglas de nomenclatura 6](#_Toc65052327)

[4.1 Interfaces 6](#_Toc65052328)

[4.2 Modelos 7](#_Toc65052329)

[5. Redux 7](#_Toc65052330)

[5.1 Nomenclatura de carpetas 7](#_Toc65052331)

[5.2 Estructura de carpetas 7](#_Toc65052332)

[5.3 Nomenclatura de ficheros 7](#_Toc65052333)

Control Documental

|  |  |  |
| --- | --- | --- |
| Redactado por: |  |  |
| Revisado por: |  |  |
| Aprobado por: |  |  |

Histórico de modificaciones

|  |  |  |  |
| --- | --- | --- | --- |
| CÓDIGO | VERSIÓN | FECHA | COMENTARIO |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Introducción

Angular es un **Framework** para aplicaciones web desarrollado en **TypeScript**, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.

La biblioteca lee el [**HTML**](https://es.wikipedia.org/wiki/HTML) que contiene atributos de las etiquetas personalizadas adicionales, entonces obedece a las directivas de los atributos personalizados, y une las piezas de entrada o salida de la página a un modelo representado por las variables estándar de JavaScript.

Angular se basa en clases tipo "**Componentes**", cuyas propiedades son las usadas para hacer el binding de los datos. En dichas clases tenemos propiedades (variables) y métodos (funciones a llamar).

Angular es la evolución de [AngularJS](https://es.wikipedia.org/wiki/AngularJS" \o "AngularJS) aunque incompatible con su predecesor.

# Reglas de arquitectura

## Organización de directorios

El proyecto, como bien sabéis, tiene la raíz en la carpeta **src** y a raíz de esta el proyecto se va a organizar de la siguiente estructura de carpetas:

* + 1. **App**
       1. **core**
       2. **login**
       3. **pages**
       4. **shared**
    2. **assets**
       1. **images**
       2. **fonts**
       3. **audios**
    3. **environments**
    4. **I18n**
    5. **scss**

**core**: En esta carpeta vamos a organizar todas las clases, constantes, pipes, decoradores, servicios, etc. que podamos utilizar de forma global en la aplicación.

**login**: En esta carpeta situamos el componente de Login debido a que se trata de un componente que esta fuera de la sección autenticada de la aplicación.

Importante: Cualquier componente que se utilice en la parte no autenticada deberá de situarse a la altura de la carpeta Login. En el caso de que sea muy frecuente, se creará una carpeta llamada **Anonymous.**

**pages:** En esta carpeta situaremos toda nuestra parte autenticada de la aplicación, por lo tanto, será la parte que más componentes albergue. Una vez dentro de la carpeta Pages, la jerarquía de las carpetas irá lo mas parecida a la estructura del menú que tengamos.

**shared**: En esta carpeta ubicaremos todos los componentes que se puedan reutilizar en mas partes de la aplicación. Dentro de la carpeta shared encontraremos dos carpetas: components y layout. Layout contiene los componentes del menú, cabecera, pie y breadcrumbs, los cuales, son comunes para toda la aplicación. El resto de componentes se guardarán en la carpeta components.

**assets**: En esta carpeta se almacenarán todos los recursos tales como, fuentes, imágenes, ficheros Excel, pdf, audios, etc.

**environments**: En esta carpeta guardaremos los ficheros de configuración. Estos ficheros guardarán toda la información relacionada a las conexiones con la API, tales como la ruta del servidor y sus diferentes métodos. También guardaremos información sensible como claves de encriptación, el tema de CSS que se quiere compilar o ejecutar.

**I18n**: En esta carpeta guardaremos todas las traducciones de los literales. La estructura interna de esta carpeta se tiene que asemejar a la estructura que tendremos en la carpeta Pages. Por lo tanto, si creamos uno componente, crearemos una nueva carpeta dentro de i18n para ese componente e intentando mantener el mismo nombre.

**scss**: En esta carpeta guardaremos todos los ficheros generales de la aplicación con los que aplicaremos estilos. No debemos de caer en el error de guardar los estilos de toda la aplicación en esta carpeta. Cada componente tendrá su propio fichero de estilos. En esta carpeta encontraremos ficheros como mixins, media queries, variables, temas, etc.

# Componentes

Explicaremos como se tienen que estructurar y nominar los componentes en función de su cometido.

## Nomenclatura de carpetas

Todas las carpetas se nombraran en minúsculas y su hubiera más de una palabra para describir la carpeta estas irían separadas por guiones ( - ). Esta nomenclatura es llamada **kebab-case**.

## Estructura de carpetas

Un componente se guardará en una sola carpeta que deberá de ser nominada con el mismo nombre que sus archivos raíz (comonent.ts, service.ts, module.ts, etc…). En el caso que subdividamos el componente en varios componentes hijos, crearíamos una carpeta llamada **components.**

Otro escenario que se presenta es cuando un componente padre tiene un componente hijo que tiene su propia ruta. En ese caso, crearíamos una carpeta **pages** para indicar que ese componente tiene un nivel de enrutado adicional y aplicaríamos todas las normas descritas en este apartado y el anterior.

En resumen, cada componente tendrá que tener creadas recursivamente carpetas para:

1. **interfaces**
2. **models**
3. **components**
4. **pages**
5. **resolvers** (en el caso de necesitar más de uno)
6. **store** (en el caso de utilizar redux)
7. **services** (en el caso de tener más de uno)

## Nomenclatura de archivos

Todos los ficheros que componen un componente, incluyendo los componentes hijos, tienen que ir nombrados en minúsculas. Si hubiera más de una palabra para nombrar un componente estas irían separadas por guiones ( - ).

## Estructura de archivos

La estructura de ficheros más habitual y que tenemos que emplear es la siguiente:

1. **component.html** para la parte visual.
2. **component.ts** para la parte de negocio.
3. **component.scss** para la parte de maquetación
4. **module.ts** para registrar todas las dependencias, servicios y configuraciones.
5. **routing.module.ts** para registrar todas las rutas a los diferentes componentes hijos.
6. **service.ts** para la parte que se encarga de la gestión de llamadas http y como complemento al negocio .ts que gestiona el negocio.
7. **guard.service.ts** para la parte de autorización (si la hubiera).
8. **resolver.service.ts** para la parte de los resolvers.

## Nomenclatura de clase

Todas las clases que creemos para conformar el componente se denominaran con nomenclatura **UpperCamelCase**. Es decir, tanto la primera palabra para nombrar la clase como las consecutivas del mismo nombre irán escritas con mayúscula.

# Reglas de nomenclatura

## Interfaces

Todas las interfaces que creemos deberán de ir guardadas en la carpeta **interfaces** correspondiente a cada componente. Las que sean de carácter general deberán de ir guardadas en la carpeta **core.**

Los ficheros se nominarán con la nomenclatura **kebab-case** y deberán de finalizar con “**.interface**.**ts”**. Por ejemplo: **pedidos-finalizados.interface.ts**

Los nombres de la interfaz (una vez dentro del fichero) deberán de nombrarse igual que el fichero, pero en nomenclatura **UpperCamelCase** y siempre irán precedidos de la letra I. Por ejemplo: **IPedidosFinalizados**

## Modelos

Todos los modelos que creemos deberán de ir guardados en la carpeta **models** correspondiente a cada componente. Las que sean de carácter general deberán de ir guardadas en la carpeta **core.**

Los ficheros se nominarán con la nomenclatura **kebab-case** y deberán de finalizar con “**.model**.**ts”**. Por ejemplo: **pedidos-finalizados.model.ts**

Los nombres del modelo (una vez dentro del fichero) deberán de nombrarse igual que el fichero, pero en nomenclatura **UpperCamelCase**. Por ejemplo: **PedidosFinalizados**

## Métodos

Todos los métodos deben de ser nominados en **lowerCamelCase** y debemos de escribir los siguientes prefijos en función de su cometido:

1. **set**: Utilizaremos la palabra “set” seguida del nombre de la acción para métodos donde guardemos algún dato en el contexto de angular
2. **get**: Utilizaremos la palabra “get” seguida del nombre de la acción para métodos donde obtengamos datos en el contexto de angular
3. **do**: Utilizaremos la palabra “do” seguida del nombre de la acción para métodos donde manipulemos datos sin guardaros en memoria.
4. **is**: Utilizaremos la palabra “is” seguida del nombre de la acción para métodos donde se validen condiciones en el contexto de angular.

# SCSS

Utilizaremos la librería Sass para la gestión de estilos de la aplicación.

## Estilos globales

Los estilos globales deberán de ubicarse en la carpeta **scss** que cuelga de la carpeta **src**. Dentro de esta carpeta encontraremos otra carpeta llamada “**temas**” donde guardaremos las diferentes sobrecargas de estilos.

Además de la carpeta temas, en la carpeta scss debemos de guardar todos los ficheros globales precedidos por el guion bajo ( \_ ). Como mínimo debemos de tener los siguientes y debemos de poder utilizarlos desde cualquier parte de la aplicación:

1. **\_botones.scss**: En este fichero almacenaremos todos los estilos de los diferentes botones.
2. **\_media**-**queries.scss**: En este fichero estableceremos los media-queries que necesitemos.
3. **\_mixins**.**scss**: En este fichero guardaremos los métodos reutilizables para el scss.
4. **\_variables**.**scss**: En este fichero guardaremos todas las variables utilizadas en nuestra aplicación.
5. **index.scss:** Será nuestro fichero de referencia para importarlo en todas nuestros ficheros de estilos
6. **themes.scss**: Únicamente utilizado para establecer los temas en función de cual se seleccione.

## Estilos de componentes

Guardaremos los estilos junto con los componentes en un archivo que se nominará igual que el componente seguido de **“component.scss”**. En este fichero guardaremos los estilos que solo serán necesarios por nuestro componente

# Redux

Redux es una librería creada por Facebook que nos permite manejar el estado de la aplicación de una manera relativamente sencilla y globalmente accesible. Le dedico una sección entera debido a que tiene un caso especial que no cumple con la totalidad de las reglas escritas anteriormente.

Además, redux puede albergar mas de un estado de aplicación al mismo tiempo.

## Nomenclatura de carpetas

Todas las carpetas de redux se llamarán **store** cuando se guarden dentro de un componente. En el caso de que se ubiquen en la carpeta **core**, debemos de utilizar la nomenclatura **kebab-case** y con el sufijo **-store.**

## Estructura de carpetas

Dependiendo de la naturaleza de cada estado, este irá ubicado en dos sitios diferentes:

1. En el caso de que el store solamente se vaya a utilizar en un componente para la gestión del mismo, deberá de ir guardado en la carpeta del propio componente en una carpeta llamada store.
2. En el caso de que el store se utilice en mas de un componente, este deberá de guardarse en la carpeta core cumpliendo con la norma de uso global.

## Nomenclatura de ficheros

Todos los ficheros que compongan el store tendrán el mismo nombre que el componente que lo utilice. En el caso de que se trate de un store que se utilice de forma global, su nombre será el de su cometido. Independientemente del escenario, se debe de utilizar la nomenclatura **kebab-case**.

Hay dos ficheros fundamentales y se deben de nominar de la siguiente manera:

1. Para el reducer escribiremos el nombre del store en **kebab-case** y añadiremos “**.reducer.ts”** al final.
2. Para las acciones escribiremos el nombre del store en **kebab-case** y añadiremos “**.actions.ts”** al final

La interfaz que utilice este store, tiene que cumplir con los estándares definidos en los puntos anteriores.