Índice

[**¿Qué son los componentes?** 1](#_Toc146789726)

[**Componentes en Angular** 1](#_Toc146789727)

[**Partes de un componente Angular** 2](#_Toc146789728)

[**Uso de Inputs** 3](#_Toc146789729)

[**Comunicando componentes** 3](#_Toc146789730)

[**Data binding en Inputs** 5](#_Toc146789731)

[**Input Set** 5](#_Toc146789732)

[**Uso de Outputs** 6](#_Toc146789733)

[**Comunicación hijo a padre** 6](#_Toc146789734)

[**Envío del mensaje** 6](#_Toc146789735)

[**Recepción del mensaje** 7](#_Toc146789736)

[**Componente para producto** 8](#_Toc146789737)

[**Comunicando componente padre a hijo** 9](#_Toc146789738)

[**Ciclo de vida de componentes** 12](#_Toc146789739)

[**Hooks más utilizados** 13](#_Toc146789740)

[**Usando hook** 14](#_Toc146789741)

[**ngDestroy and SetInput** 16](#_Toc146789742)

[**Liberando espacio de memoria** 16](#_Toc146789743)

[**RxJS** 16](#_Toc146789744)

[**Servicios** 18](#_Toc146789745)

[**Comando para generar un servicio** 18](#_Toc146789746)

[**Archivos generados** 19](#_Toc146789747)

[**Como utilizar los servicios** 19](#_Toc146789748)

[**Inyección de dependencias** 19](#_Toc146789749)

[**Patrón Singleton** 20](#_Toc146789750)

[**Peticiones HTTP con servicios** 21](#_Toc146789751)

[**Tipar los datos que devuelve una petición** 23](#_Toc146789752)

[**Pipes** 23](#_Toc146789753)

[**Construir un Pipe propio** 24](#_Toc146789754)

[**Directivas** 25](#_Toc146789755)

[**Comando para generar una Directiva** 26](#_Toc146789756)

[**Estructura interna de una directiva** 26](#_Toc146789757)

[**Reactividad básica** 27](#_Toc146789758)

[**State management** 27](#_Toc146789759)

[**¿Qué pasa si el componente hijo quiere comunicarse con el componente root?** 27](#_Toc146789760)

[**¿De qué trata esta teoría?** 28](#_Toc146789761)

[**Estilos y Linters en Angular** 30](#_Toc146789762)

# **¿Qué son los componentes?**

**Un componente** es una pieza de software con una **responsabilidad única** y una **estructura y funcionalidad determinada**, además de ser **reutilizable**.

Es una manera de dividir tu aplicación de una forma escalable para no tener todo en un solo archivo. Por ejemplo, un componente para el **header**, otro para el **footer**, uno más para el **menú**, etc.

## **Componentes en Angular**

Puedes crear tu primer componente en Angular utilizando el comando **ng generate component test-name** o en su forma corta **ng g c test-name**.

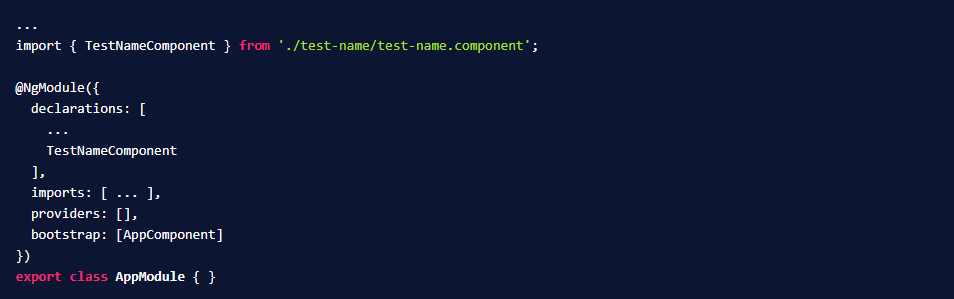
Esta acción creará los siguientes archivos:

* my-test-name.component.html
* my-test-name.component.ts
* my-test-name.component.css
* my-test-name.component.spec.ts

1. El archivo **.html** que será el template que tu componente utilizará.
2. El archivo **.ts** que contiene el código TypeScript y la lógica.
3. El archivo **.css** que contiene los estilos.
4. Si escogiste trabajar con un preprocesador de CSS, este archivo puede ser **.scss**, **.sass** o **.less**.

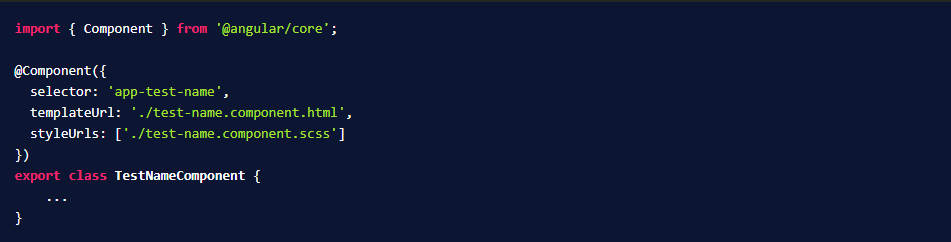
Finalmente, el archivo más extraño, **.spec.ts** que contiene el código de las pruebas unitarias que puedes escribir para automatizar el testing en tu componente.

Angular también importará automáticamente el componente creado en el archivo **app.module.ts** para que automáticamente puedas utilizarlo en tu aplicación.



# **Partes de un componente Angular**

El archivo con la extensión **.ts** es el componente principal de cualquier Angular.



Observa lo más importante, el decorador **@Component()**.

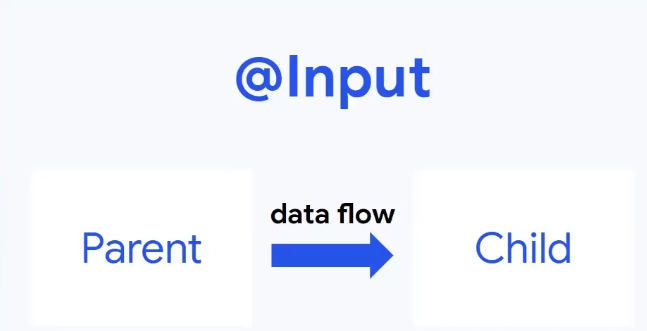
Los decoradores alteran el comportamiento de una clase en Angular, para que el compilador de TypeScript interprete el código de la manera correcta y sepa que una clase es:

* Un componente.
* Un módulo.
* Un servicio.
* Una directiva, etc.

Este decorador es quién enlaza el componente con el archivo HTML y la hoja de estilos, además le otorga al componente un selector o un nombre para utilizarlo en tus templates.

# **Uso de Inputs**

Para comunicar componentes, Angular hace uso de decoradores para intercambiar información entre un **componente padre** hacia un **componente hijo** y viceversa.



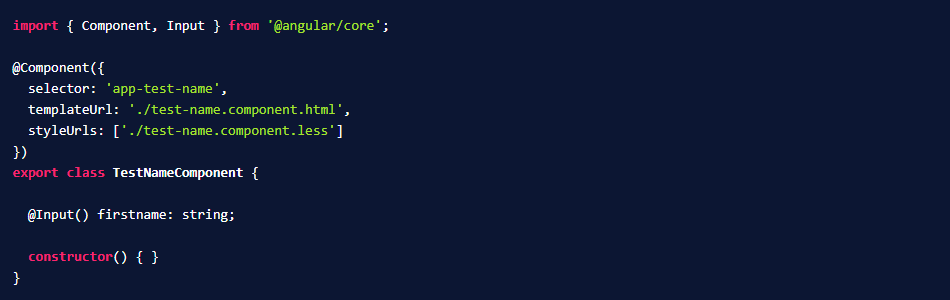
## **Comunicando componentes**

Para enviar información de **padre** a **hijo**, puedes utilizar el decorador **@Input()** para marcar una propiedad de una clase como punto de entrada de un dato.

**<p style=‘text-align:center;’>**

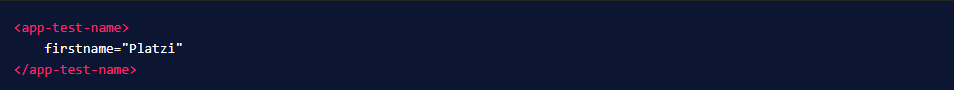
**<img src=“**https://static.platzi.com/media/articlases/Images/Screenshot from 2022-04-05 22-42-58.png**” alt=“**Envio de datos componente padre a hijo**”>**

**</p>**



Debes importar **Input** desde **@angular/core** para poder utilizar esta directiva e indicar que la propiedad **firstname** es un dato que el **componente padre** enviará.

Podrás inicializar el componente desde su padre y pasarle los inputs que este necesite de la siguiente manera:



También puedes cambiar el nombre del **Input** especificando el nombre de la propiedad que quieras que este utilice al inicializar el componente.





## **Data binding en Inputs**

El decorador **@Input()** detectará cualquier cambio en el dato y automáticamente actualizará su valor. Si ocurre algún evento en el componente padre que cambie el valor en el Input **firstname**, el componente hijo recibirá inmediatamente ese nuevo valor.

## **Input Set**

Otra manera de utilizar la directiva **@Input** es de la siguiente manera:



Observa que en esta oportunidad, cada vez que se envía un valor al **@Input**, se ejecutará la función **saludar()** que recibe como parámetro el valor que se le haya enviado.

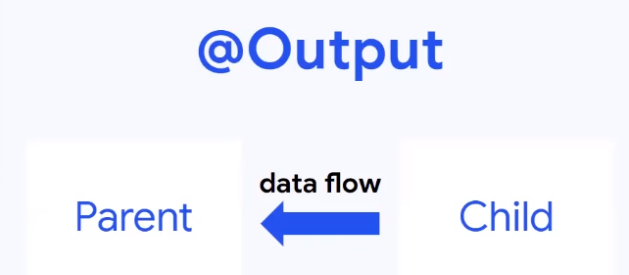
De esta manera, puedes ejecutar la lógica que necesites dentro de esta función cada vez que el valor del **@Input** cambia.

# **Uso de Outputs**

Así como el decorador **@Input** permite el envío de información desde un **componente padre** hacia un **componente hijo**, el uso de **@Outputs** permite lo contrario.

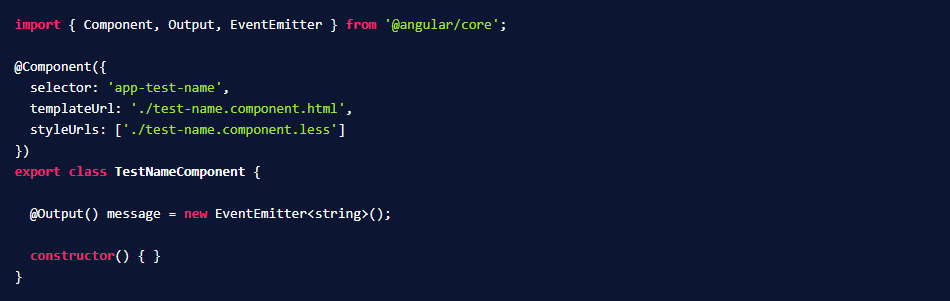
## **Comunicación hijo a padre**

A partir de la emisión de un evento, el decorador **@Output()** permite enviar mensajes desde un **componente hijo** hacia **el padre**.



## **Envío del mensaje**

Para esto, se hace uso de la clase **EventEmitter** importándola desde **@angular/core**, para crear en tu componente una propiedad emisora de eventos.



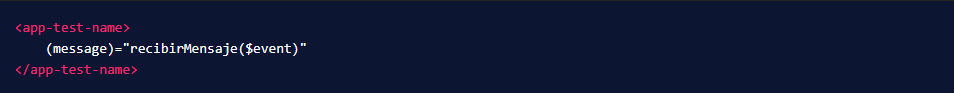
Decorando la propiedad con el **@Output()** y creando una instancia de **EventEmitter** podrás emitir un evento de la siguiente manera:



Llamando al método **emit()** de la instancia **EventEmitter**, se enviará el valor al **componente padre** que se encuentre escuchando el evento.

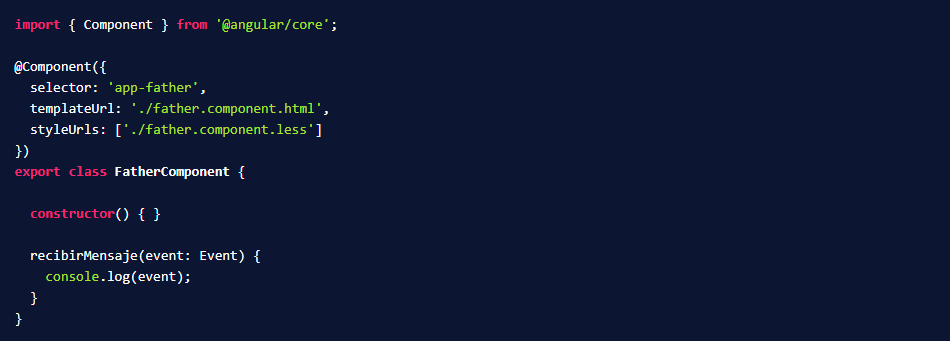
## **Recepción del mensaje**

Desde el **componente padre**, inicializa el **componente hijo** de la siguiente manera:



Se “**bindea**” la propiedad emisora de eventos con **()** y se le pasa una función que se ejecutará cada vez que emita un evento.

Y en el componente padre:



La función **recibirMensaje()** posee un parámetro del tipo **Event** que contendrá el mensaje del componente hijo.

# **Componente para producto**

Existen muchas situaciones en donde deberás enviar información de un componente padre a su/s hijo/s, por eso, acá te mostraremos con un ejemplo cómo funciona el componente para producto.

## **Comunicando componente padre a hijo**

Un ejemplo real para el uso de la comunicación entre componente podría ser para renderizar N cantidad de productos de un catálogo.

**Paso 1**: Comienza creando una interfaz para tipear el modelo de datos del Producto:



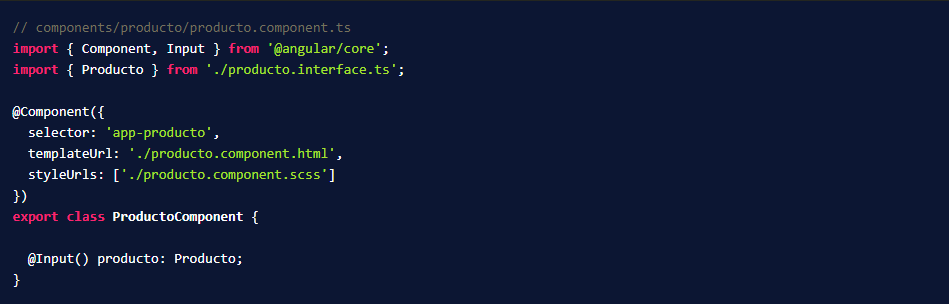
**Paso 2**: Luego, impórtala en el componente Catálogo que será el componente padre en la comunicación.



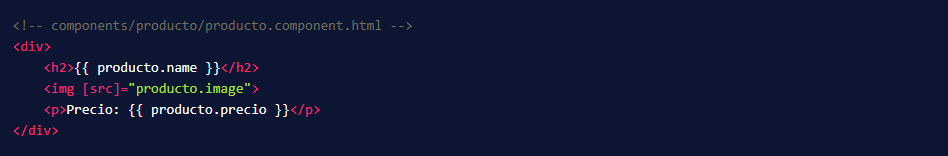
**Paso 3**: Este componente posee un array de productos para iterar en el HTML inicializando el componente <app-producto> por cada objeto en el array.



**Paso 4**: Finalmente, el componente hijo recibe el producto haciendo uso del decorador @Input() y apoyándose también de la interfaz para tipear los datos.



Pudiendo mostrar la información del producto en el template del componente hijo:

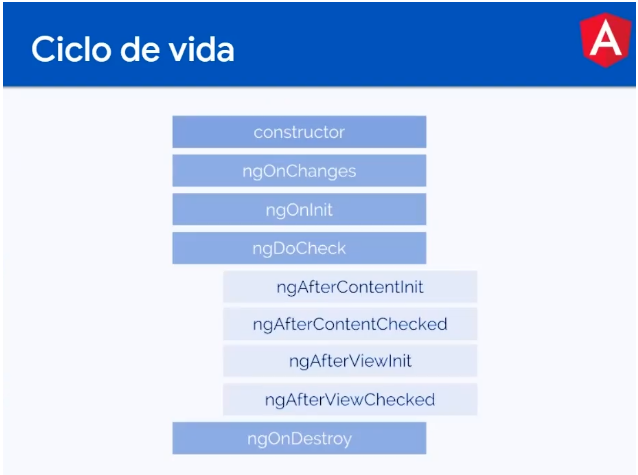


Será habitual tener la necesidad en tus proyectos de construir componentes más grandes o “contenedores” de muchos otros componentes repetitivos y más pequeños. Es importante buscar este desacople entre componentes de la mejor manera posible.

# **Ciclo de vida de componentes**

**Un componente pasa por varias etapas en su ciclo de vida**. A través de hooks, **puedes realizar una determinada acción** cuando el componente es inicializado, cuando se dispara un evento, cuando se detecta un cambio, cuando el componente es destruido, etc.

A continuación, se detalla la secuencia de eventos y el orden de los mismos:



## **Hooks más utilizados**

**Constructor**

Como en toda clase en la programación orientada a objetos, el **constructor** es quien crea la instancia del objeto y sus dependencias.

* Solo se ejecuta una vez antes del render del componente.
* No tiene que utilizarse para procesos asincrónicos.

**ngOnChanges**

El hook ngOnChanges() se dispara cada vez que se produce un cambio de estado en el componente. Cuando una variable cambia de valor, por ejemplo o ante el cambio de valor de un Input.

* Se ejecuta N cantidad de veces antes y durante el render del componente.
* Puede emplearse para procesos asincrónicos.

**ngOnInit**

Es el hook más usado, ngOnInit() es ideal para cualquier solicitud de datos asincrónicos a una API para preparar el componente antes de renderizarlo.

* Únicamente se ejecuta una vez, antes del render del componente.
* Puede usarse para procesos asincrónicos.

**ngAfterViewInit**

Este hook únicamente se ejecuta una vez cuando el render del componente haya finalizado. Puede serte útil para realizar acciones programáticas que requieran que todo el HTML del componente ya este preparado.

* Únicamente se ejecuta una vez después del render del componente.

**ngOnDestroy**

Finalmente, **ngOnDestroy()** se ejecutará cuando el componente es destruido, o sea, cuando ya no existe en la interfaz del navegador. Suele utilizarse para liberar espacios de memoria que el componente requiera.

## **Usando hook**

Los hooks de ciclo de vida de Angular, son interfaces que tienen que importarse desde @angular/core para implementarlos en la clase y así detectar los cambios en cada evento.



Cada hook tiene sus características y utilidades recomendadas dependiendo lo que necesitas. Es importante seguir estas recomendaciones para buscar optimizar el rendimiento de tu aplicación.

# **ngDestroy and SetInput**

El hook **ngOnDestroy()** & **SetInput** tiene una importancia clave para el cuidado de nuestra aplicación. **Su funcionalidad más importante es la liberación de espacio en memoria** de variables para que no se acumule. Si esto llegara a suceder en tu aplicación, la misma podría volverse lenta y tosca a medida que toda la memoria del navegador es ocupada.

## **Liberando espacio de memoria**

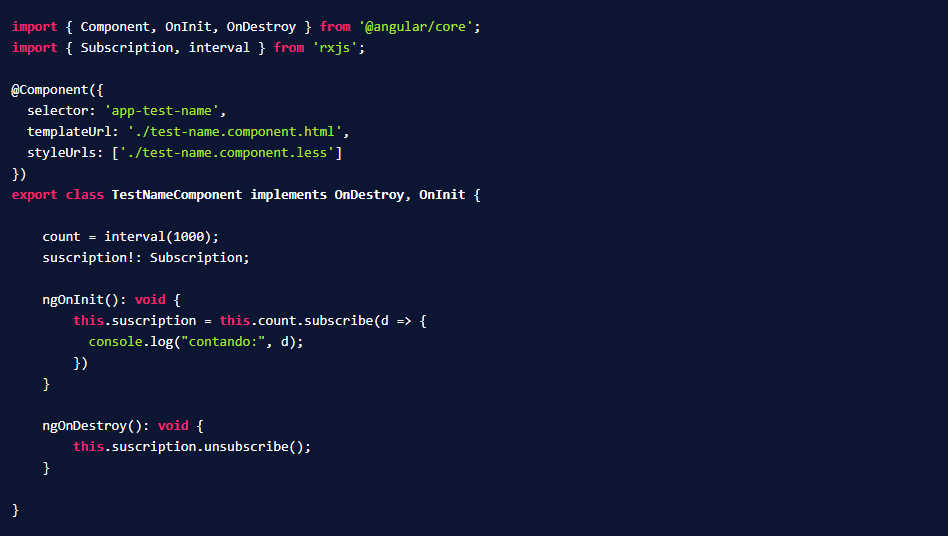
Todo el ecosistema Angular está basado en observables para el manejo asincrónico.

Cada vez que utilices un **subscribe()** para escuchar la respuesta de algún evento asincrónico (por ejemplo, el llamado a una API), es relevante realizar el respectivo **unsubscribe()** para liberar ese espacio en memoria.

## **RxJS**

[RxJS](https://rxjs.dev/) (Reactive Extensions Library for JavaScript) es una popular librería de Javascript para el manejo de observables. Si trabajas con Angular esta librería será tu mejor amiga.

Observa el siguiente ejemplo donde primero se importa **Subscription** desde **rxjs** para tipar la variable **suscription**. Guardamos el observable para posteriormente darlo de baja. También importamos **interval** que devuelve el observable y genera un contador que emite una pulsación, en este ejemplo, cada 1000 milisegundos.



En el **ngOnInit()**, se está suscribiendo a la propiedad **this.count** para imprimir por consola, cada 1000 milisegundos, el contador. Podrás observar en la consola del navegador el contador corriendo:



Si nuestro código acabara aquí, cuando el componente es destruido, el contador continuaría ocupando memoria que ya no debería ser utilizada.

Para solucionar esto, guardamos en **this.suscription** el observable del contador y en **ngOnDestroy()** y llamamos al método **.unsubscribe()** para detener el contador.

# **Servicios**

Es la forma en que Angular nos permite hacer modular nuestra aplicación y apartar toda nuestra lógica de negocio, (todo lo que no tiene que ver con la UI, como manipular datos o hacer servicios compartidos).

Los servicios únicamente comunican 'Lógica de negocio' hacia los componentes y estos pueden ser utilizados a través de toda la aplicación, por los todos componentes.

## **Comando para generar un servicio**

* Version Larga: **ng generate service <nombre-service>**
* Version abreviada: **ng g s <nombre-service>**

Es buena práctica guardar todos nuestros servicios en una carpeta llamada **services**

**NOTA**: Los servicios pueden utilizarse dentro de los componentes y/o también dentro de otros servicios.

## **Archivos generados**

Angular nos genera dos archivos, un archivo **.ts** donde estará la lógica y, un archivo **.spec.ts**, que puede utilizarse para tests

## **Como utilizar los servicios**

Para utilizar un servicio debemos de seguir estos pasos:

1. Importar el servicio en el componente a ser utilizado

**import { StoreService } from '../../services/store.service';**

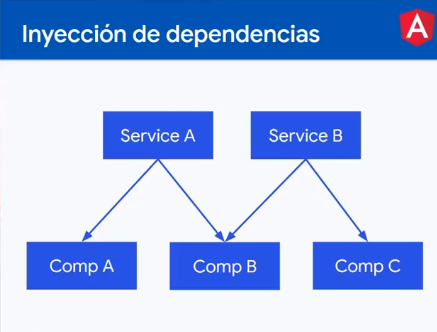
1. Para incluir a este servicio dentro del componente, vamos a crear algo llamado **Inyección de dependencias**

Dentro del constructor, instanciamos una variable, y le decimos que esta variable, del tipo de servicio que vayamos a utilizar.



# **Inyección de dependencias**

El decorador **@Injectable**

El decorador de un servicio es **@Injectable**, el cual permite que el servicio se pueda 'Inyectar' en otros componentes y servicios.

Dentro del decorador, está declarado el Alcance o Scope del servicio.



El motor de inyección de dependencias de Angular funciona de la siguiente manera:

* Nuestro servicio es 'Inyectado' dentro de un componente.



Aquí tipamos la variable al tipo de servicio que vayamos a utilizar, pero no la instanciamos, ya que el motor de Inyecciones hace esto por nosotros.

Desde ahora, cuando nuestro componente llame a la variable tipada con el tipo de servicio, el motor de inyecciones se encargará de crear la instancia por nosotros.

Pero, el motor de inyecciones también hace algo más por nosotros.

## **Patrón Singleton**

Qué pasa si hay dos componentes utilizando el mismo servicio. ¿Se crean dos instancias del servicio?

No. Aquí el motor de inyecciones de Angular emplea el patrón Singleton

Esto lo que hace es, guardar en memoria la instancia creada, y devuelve la referencia a todos los componentes que la necesiten. Con eso no creamos instancias por cada componente que este requiriendo ese servicio.

# **Peticiones HTTP con servicios**

Angular cuenta con un módulo específico para las peticiones HTTP.

1. Crear el servicio encargado de realizar las peticiones HTTP.
2. Importar el Modulo **HttpClientModule** en el **app.module** de nuestra aplicación.
3. Agregar el **HttpClientModule** a la sección de **imports**
4. Vamos a nuestro servicio e importamos el SERVICIO **HttpClient**
5. Como el HttpClient es un servicio, debemos de declararlo en el constructor, como cualquier otro servicio
6. Realizamos la petición a nuestra url con ayuda de **HttpClient.get**



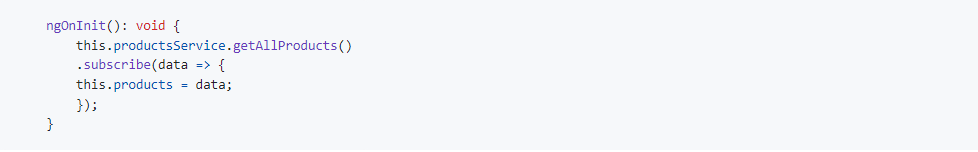
1. Debemos ir a nuestro componente y crear la **Inyección dependencias** para poder utilizar el servicio.



1. Como una petición **Http** es una operación asíncrona, no podemos hacerla desde el constructor. Recordemos que un lugar apropiado para utilizar funcionalidades asíncronas es en el **ngOnInit**.



1. Angular maneja un formato por defecto para los resultados asíncronos, un **Observable**. Podemos acceder a los datos de este Observable utilizando el método **subscribe**.



## **Tipar los datos que devuelve una petición**

Esto lo hacemos para asegurarnos que la petición Http nos devuelve los datos con el formato que esperamos.



# **Pipes**

Los pipes funcionan como una tubería, tienen una **entrada**, por donde entran los datos, posteriormente se **procesan esos datos** y nos produce una **salida**. Estos son utilizados desde el HTML únicamente.

Una característica de los pipes es que se pueden unir.



En este ejemplo, tenemos la manera de utilizar un pipe:

* Nuestra variable **date\_var** constituye la entrada del pipe.

**<p> Fecha: {{ date\_var }}</p>**

* El símbolo **|** es el que utilizamos para 'decir' que vamos a utilizar un **Pipe**

**<p> Fecha: {{ date\_var | }}</p>**

* Luego le sigue el nombre del Pipe.

**<p> Fecha: {{ date\_var | date }}</p>**

* Y por último tenemos el formato que queremos darle a nuestra salida

**<p> Fecha: {{ date\_var | date:'short' }}</p>**

# **Construir un Pipe propio**

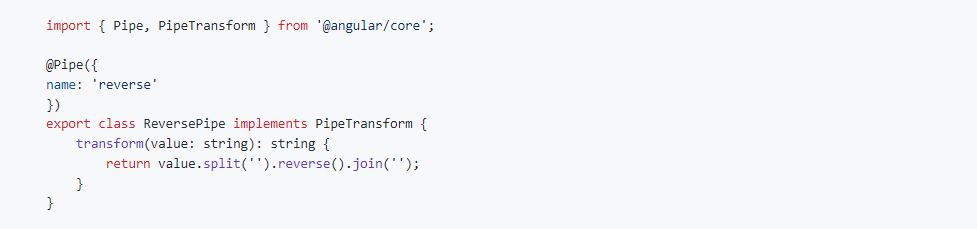
Comando para generar un Pipe

* **Version Larga**: ng generate pipe nombre-pipe
* **Version abreviada**: ng g p nombre-pipe

Angular nos genera dos archivos, un archivo .ts donde estara la logica y, un archivo .spec.ts, que puede utilizarse para tests

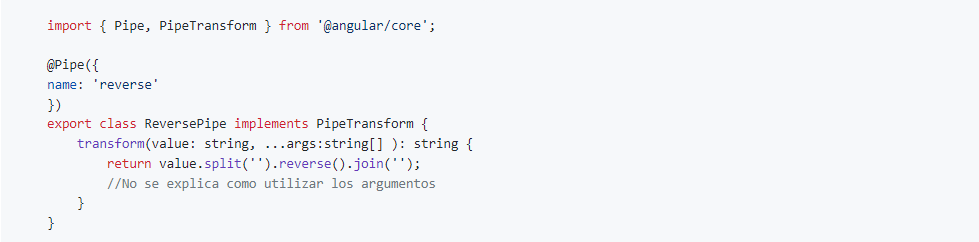
Estructura interna de un pipe

1. Simple



* El parámetro **value** constituye la entrada del Pipe.
* Luego simplemente retornamos el valor con el cambio deseado.

1. Con argumentos



# **Directivas**

Se utilizan para hacer modificaciones del DOM de forma directa y también, podemos modificar atributos.

Normalmente evitamos hacer modificaciones del DOM, porque los frameworks ya hacen esto por nosotros.

## **Comando para generar una Directiva**

* **Version Larga**: ng generate directive nombre-directive
* **Version abreviada**: ng g d nombre-directive

Es buena práctica guardar las directivas dentro de una carpeta llamada **directives**

Angular nos genera dos archivos, un archivo **.ts** donde estará la lógica y, un archivo **.spec.ts**, que puede utilizarse para tests

## **Estructura interna de una directiva**



Con ayuda del servicio **ElementRef**, nos traemos una referencia del elemento HTML y desde ahí podemos modificarlo.

Para utilizar esta directiva simplemente debemos de ponerlo como un atributo dentro del elemento HTML que queremos modificar.



# **Reactividad básica**

## **State management**

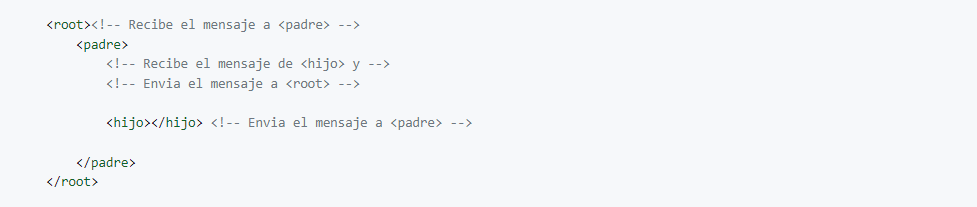
Para esto, debemos de tener en cuenta que nuestros componentes forman un árbol por jerarquías.

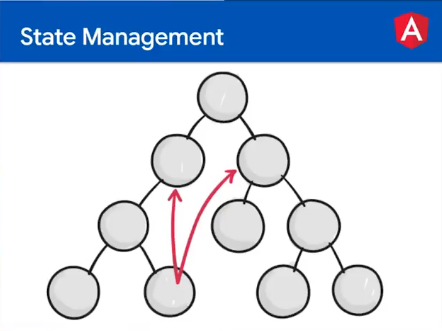
Por ejemplo: Si tenemos al componente **root**, que dentro tiene un componente **padre** y a su vez, este componente padre tiene un componente **hijo**.

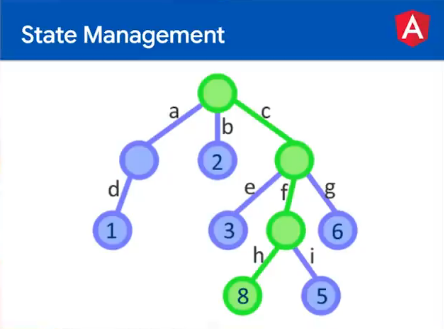


## **¿Qué pasa si el componente hijo quiere comunicarse con el componente root?**

Si seguimos el estándar de Angular (**@Inputs** y **@Outputs**), cada componente tendría un **@Output** hasta llegar al **root**.







Para evitar tener que hacer este recorrido de componentes, se han creado estrategias para poder comunicarse de un componente a otro sin tener que hacer todo el recorrido.

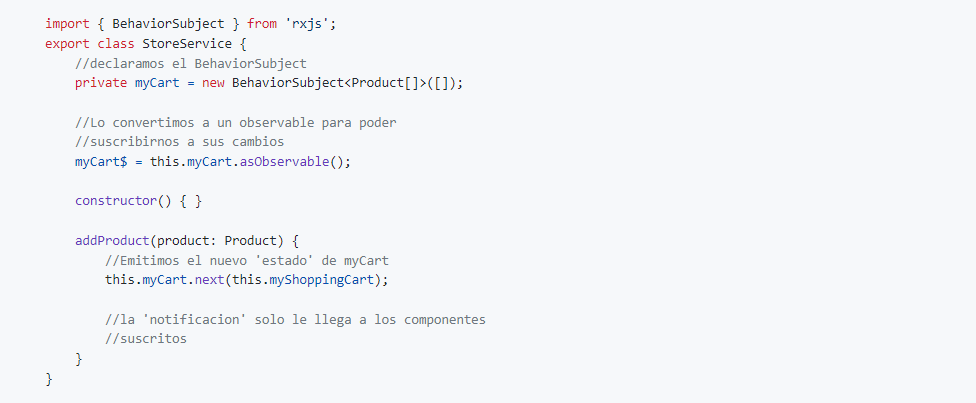
## **¿De qué trata esta teoría?**

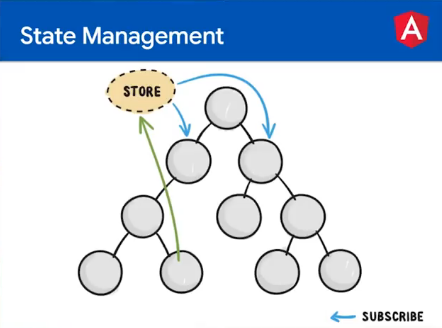
Se trata de tener un **'Store'** o un estado global de la aplicación, en donde, en lugar de recorrer cada componente para poder comunicarme, debo crear un **'Store'** donde se almacenen los estados que se van a compartir.

De esta forma, los componentes que quieran ese estado, simplemente se suscriben.

**Como Aplicarlo a código**

Para esto, creamos un servicio, el cual utilizaremos como **'Store'** y, como variable para almacenar los estados utilizaremos un **BehaviorSubject** de **rxjs**.





**Como escuchar los cambios desde el componente**

1. Inyectamos el servicio en el componente.
2. Nos suscribimos desde el ngOnInit o desde el constructor.



Buenas prácticas para definir **observables**

A los observables, normalmente se les agrega un signo de pesos (**$**) al final **myCart\$**

# **Estilos y Linters en Angular**

Aplicar las buenas prácticas de forma sistematizada

Antes que nada, angular cuenta con una guía de buenas prácticas en su [página oficial](https://angular.io/guide/styleguide#naming)

Para ver si cumples con estas buenas practicas puedes utilizar un Linter

**Pasos**

* Configurar el linter **ng add @angular-eslint/schematics**
* Correr el linter **ng lint**