Directivas: son una forma de agregar nuevas capacidades a un elemento de su pantalla.

https://lemoncode.net/lemoncode-blog/2017/12/12/git-y-visual-studio-code

Templates: Especifican como se mostraran los componentes.

Cosas a recordar de Angular:

Faster

Dependency Injection

Zone.js

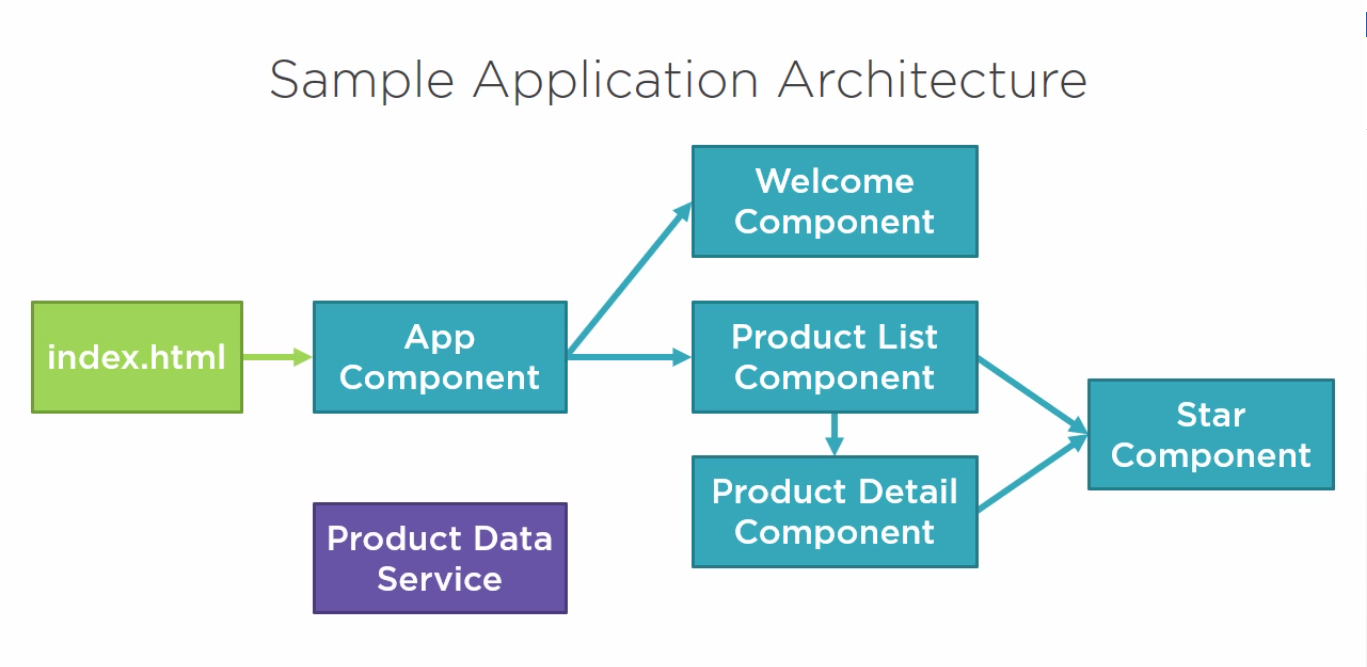
Multiple Rendering Targets

Que es un componente?

Pre-requisitos

http://blogs.msmvps.com/deborahk/angular-2-getting-started-problem-solver/

https://github.com./Deborahk/Angular-GettingStarted



1. **First Things First**



**TypeScript**: es un lenguaje de código abierto que es un supe conjunto de JavaScript y se compila con JavaScript simple mediante la compilación. Está fuertemente tipado, por lo que cada variable, cada función y cada parámetro de función pueden tener un tipo de datos.

Implemente clases, interfaces y herencia.

**TypeScript Playground**

<http://www.typescriptlang.org/Playground/>

**Setting up Our Environment**

1 – Install NPM

2 – Set up the Angular Application

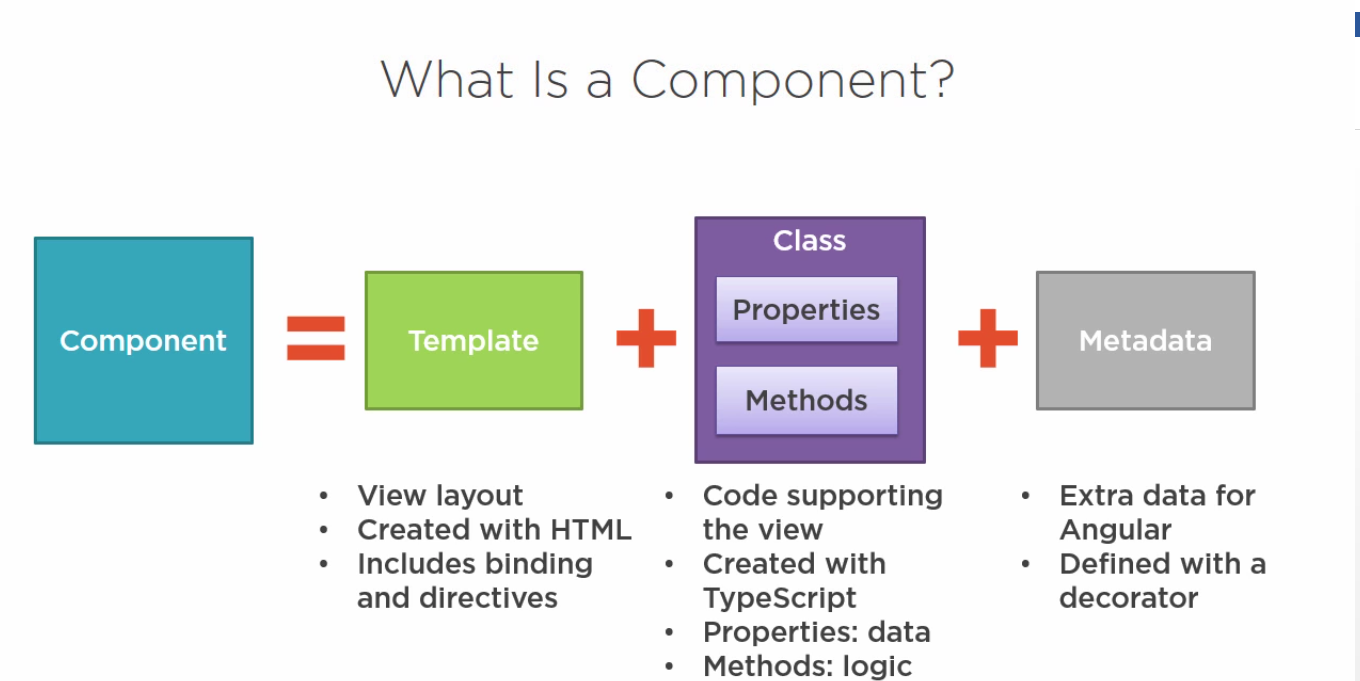
Los módulos ayudan a organizar la aplicación en bloques cohesivos de funcionalidad. Cada aplicación posee por lo menos un módulo Angular.

Podemos definir módulos compartidos, estos sería bueno si tenemos dos módulos separados pero entre ellos usan una función idéntica, podríamos escribir un módulo compartido para utilizar entre ambos. (Herencia)

Los módulos están segmentados a su vez en Componentes, y los componentes solo pueden pertenecer a un solo Modulo.

1. **Introduction to Components**





Una clase se convierte en Componente cuando le damos metadatos.

Angular necesita metadatos para comprender como crear una instancia del componente, construir la vista e interactuar con el componente.

Un **Decorator** (Decorador) es una función que agrega metadatos a una clase, sus miembros o sus argumentos. A su vez es una función JavaScript que se implementa en TypeScript.

El Decorator siempre tiene el prefijo **@**.

Como el Decorator es una función colocamos luego del Nombre los () EJ:

@Component

( {

} )

Esta Sintaxis es diferente en algunas cosas, tal como el uso del ; no es necesario en los Decorator

El **Selector**: Define el nombre directivo del Componente.

Una **Directiva** es simplemente una etiqueta HTML Personalizada.

Un componente siempre debe de tener una **Plantilla** o **Template.**

Con los templetes Definimos el diseño para el Fragmento de la interfaz de usuario o la vista administrada por este componente. 

Las dobles llaves {{ }} indican la unión de datos (Data Binding)

Para ver los paquetes que podemos importar en Angular

<https://www.npmjs.com/~angular>

3 – **Templates, Interpolation, and Directives**



<https://getbootstrap.com/>

<https://fontawesome.com/>

Para Instalar Bootstrap y Font-awesome al proyecto debemos de correr en la terminal el comando:

**Npm install bootstrap font-awesome**

Y luego en el archivo de styles.css agregar estas dos líneas:

@import "~bootstrap/dist/css/bootstrap.min.css";

@import "~font-awesome/css/font-awesome.min.css";

**Binding** es quien coordina la comunicación entre la clase del componente con el Template (Plantilla) y a menudo implica el paso de datos.

A las Dobles {{ }} se le llama Interpolation.

La interpolación (Interpolation) es un enlace unidireccional, desde la propiedad de la clase a la Plantilla (Template).

La interpolación soporta: Concatenación, cálculos simples.

Las Directivas (Directives) es como un elemento o atributo HTML personalizado que utilizamos para potenciar y ampliar nuestro HTML.

Podemos crear nuestras propias Directivas o utilizar las directivas integradas de Angular.

Directivas Estructurales: **\*ngIf: If logic** and **\*ngFor: For loops**

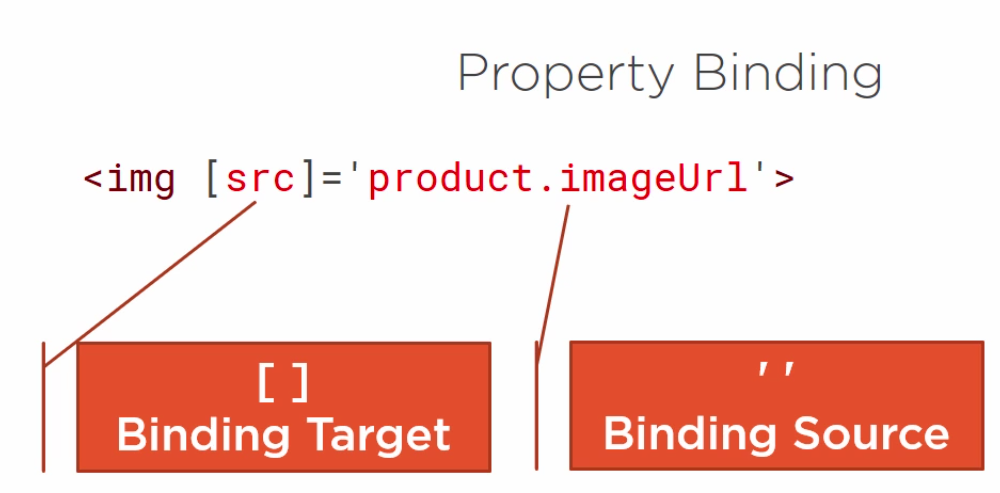
El \* delante nos indica que es una Directiva Estructural.

\*ngIf es una directiva estructural que elimina o recrea una parte del árbol DOM en función de una expresión.

4 – **Data Binding & Pipes**



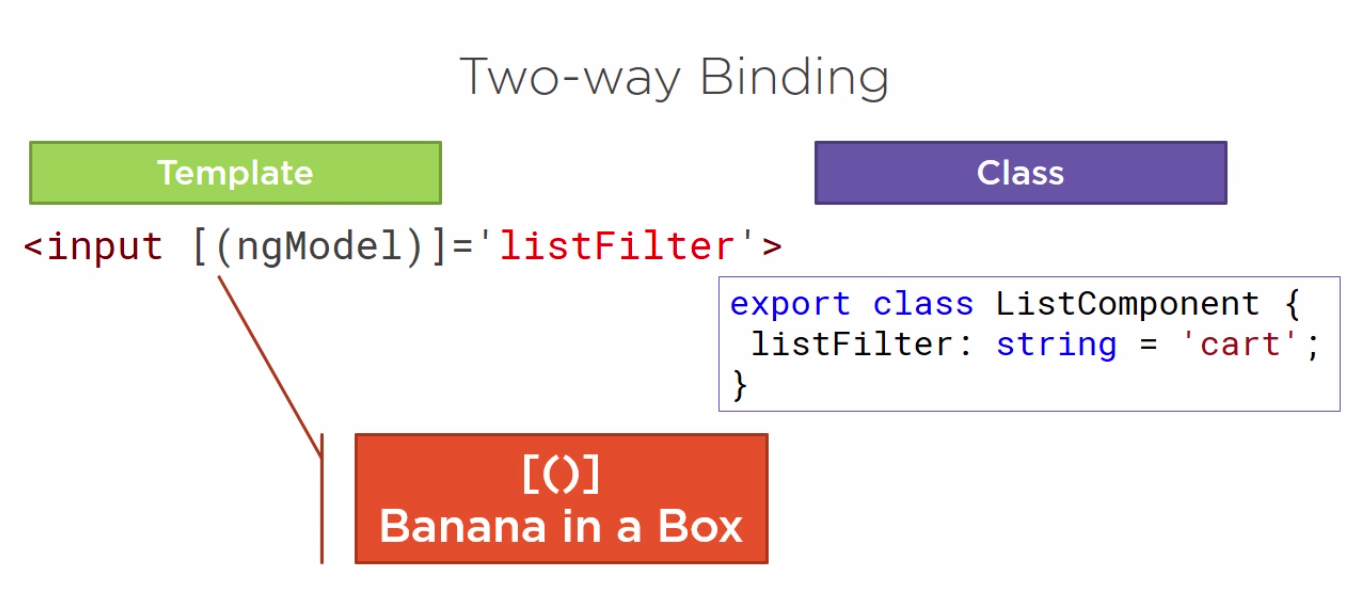
**El Property Binding (Enlace de Propiedad)** nos permite establecer una propiedad de un elemento en el valor de una expresión de plantilla.



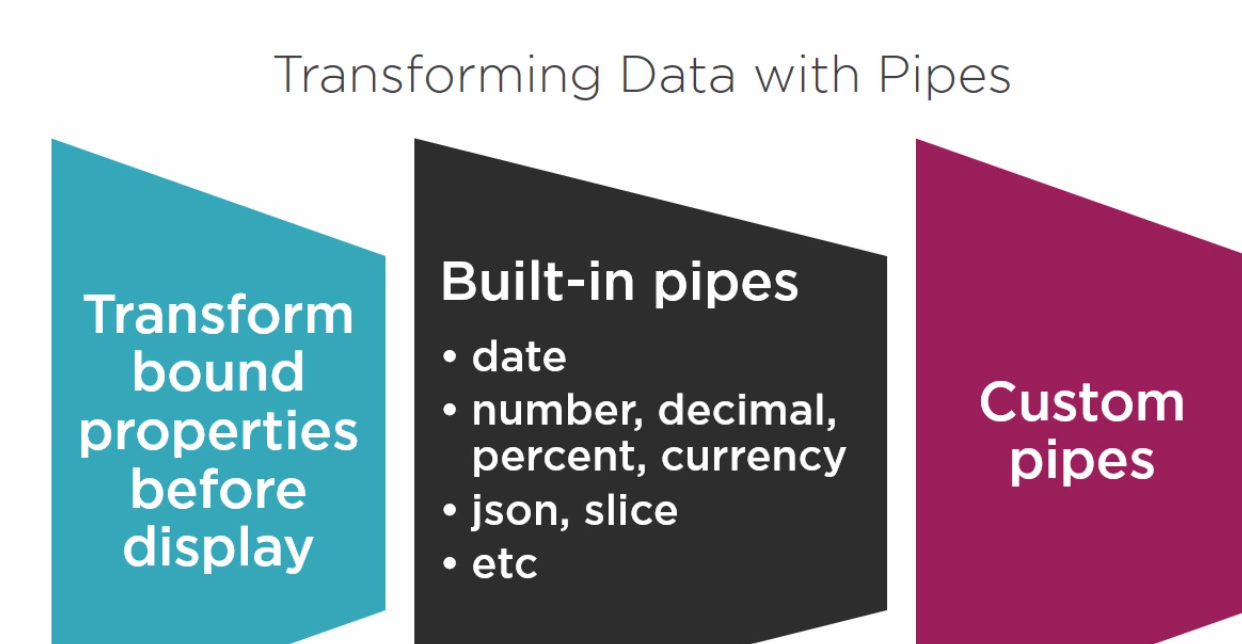
**Event Binding**

<https://developer.mozilla.org/en-US/docs/Web/Events>

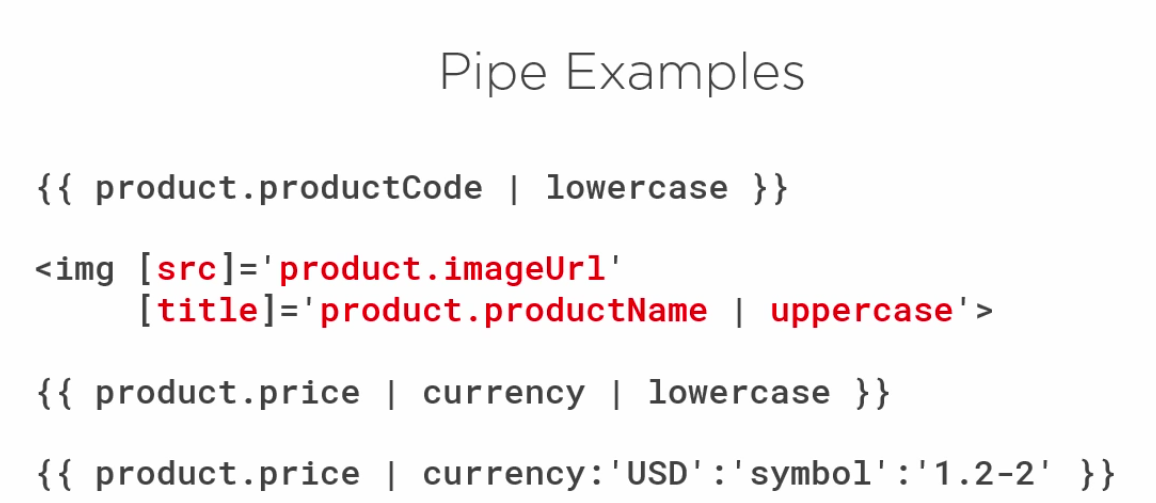
Para crear especificar un enlace Bidireccional (Two-way Binding) se utiliza la directiva **ngModel** entre corchetes**.**

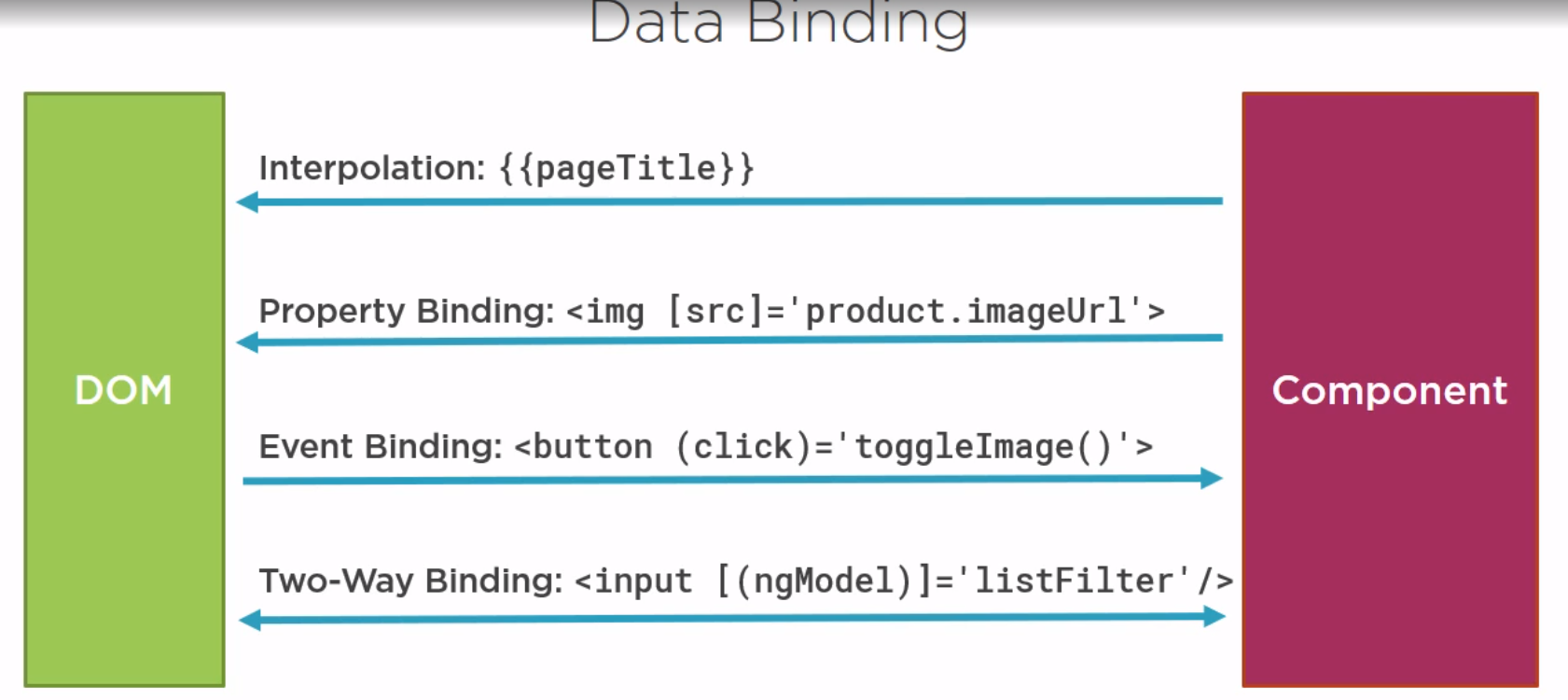


PIPES



Pipes Ejemplos:





**7 – More o Components**



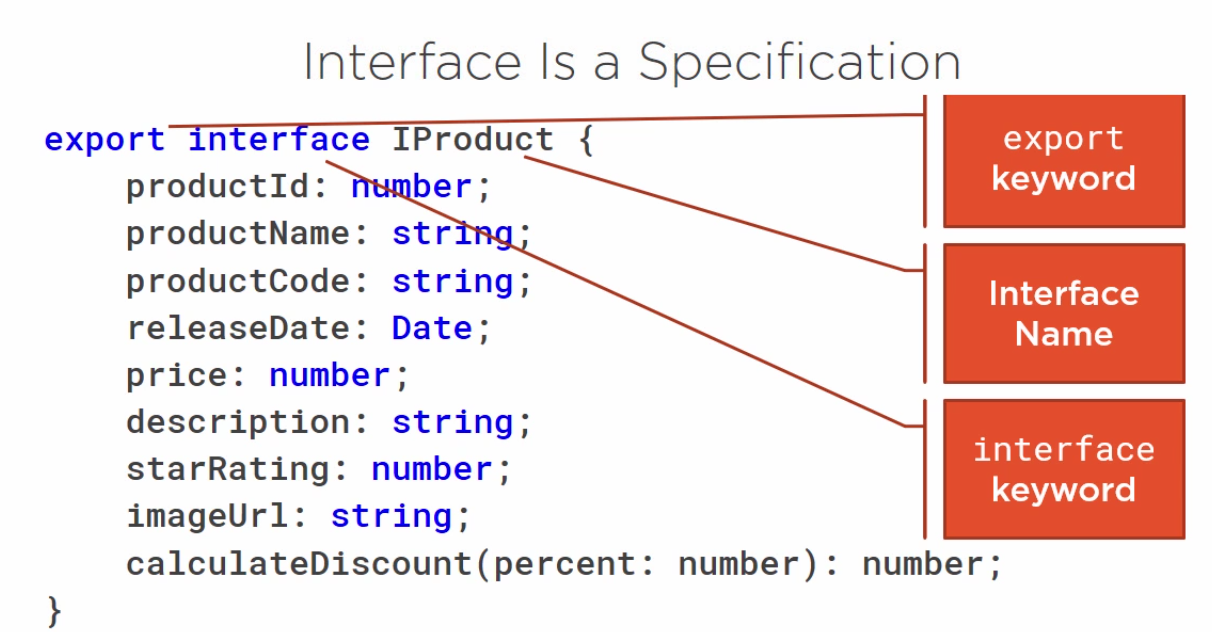
**Interface:** es una especificación que identifica un conjunto relacionado de propiedades y métodos.

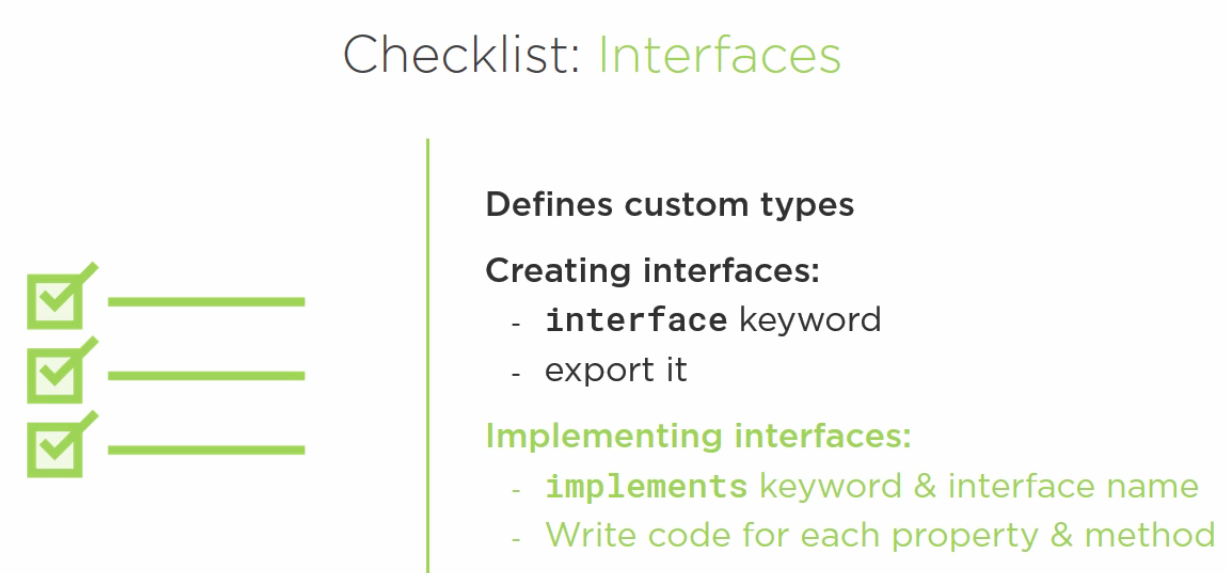
Una clase se compromete a respaldar la especificación implementando la interfaz. Eso significa que la clase incluye código para cada propiedad y método identificado en la interfaz.

Entonces podemos usar la Interfaz como un tipo de datos.

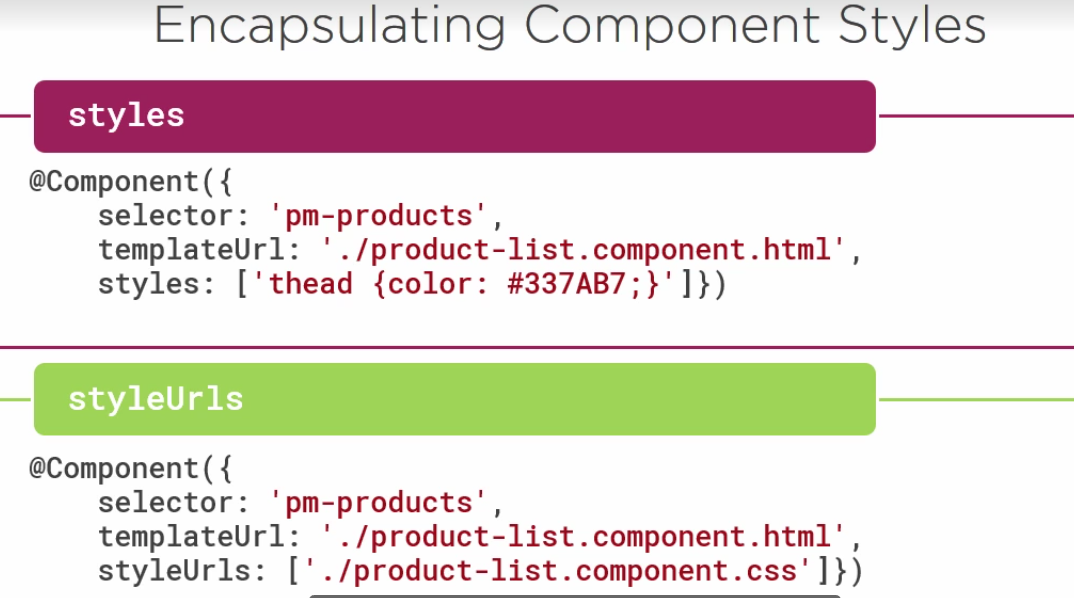
Las interfaces son solo en tiempo de Desarrollo (Development Time Only).

Su propósito es proporcionar una tipificación sólida y un mejor soporte de herramienta a medida que construimos y recreamos nuestro código.





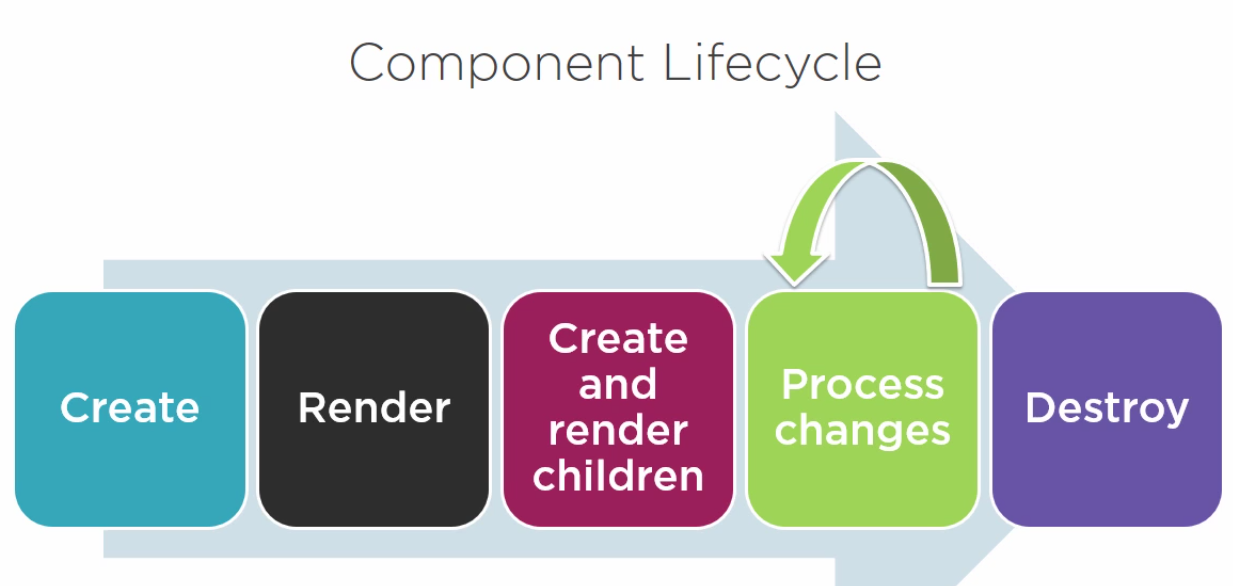
Utilizamos la I para identificar las interfaces y luego la implantamos en el archivo .ts IProdutc[]





**Ciclo de Vida de un Componente:**

1. Angular Crea el Componente
2. Lo Procesa
3. Crea y Procesa los Hijos
4. Procesa los cambios
5. Lo destruye antes de eliminar su plantilla del DOM.



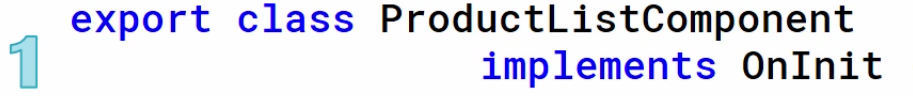
**Lifecycle hooks:** Angular nos proporciona algunas herramientas para que tengamos control del ciclo de vida de los componentes.

**Component Lifecycle Hooks:**

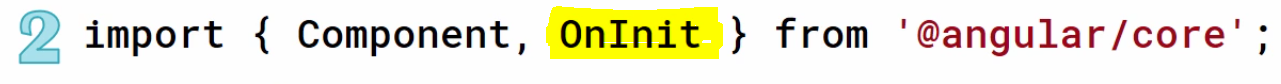
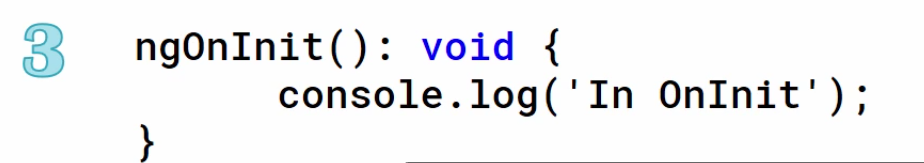
1. **OnInit**: Para realizar la inicialización de Cualquier componente después que Angular haya Inicializado las propiedades de Enlace de Datos.
2. **OnChanges**: para realizar cualquier acción después que Angular establezca las propiedades de entrada enlazadas a Datos.
3. **OnDestroy**: Para realizar cualquier limpieza antes de que Angular destruya el componente.

Para utilizar un enlace de ciclo de Vida. Implementamos la Interfaz de enlace de ciclo de vida.

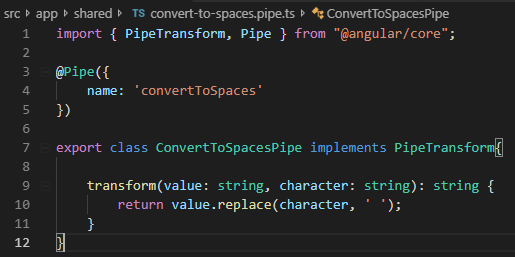
En primer lugar debemos de implementar nuestra interfaz del ciclo de Vida:



Luego debemos de importar la interfaz que utilizaremos:

El tercer paso es implementar utilizar el método de la interfaz: 

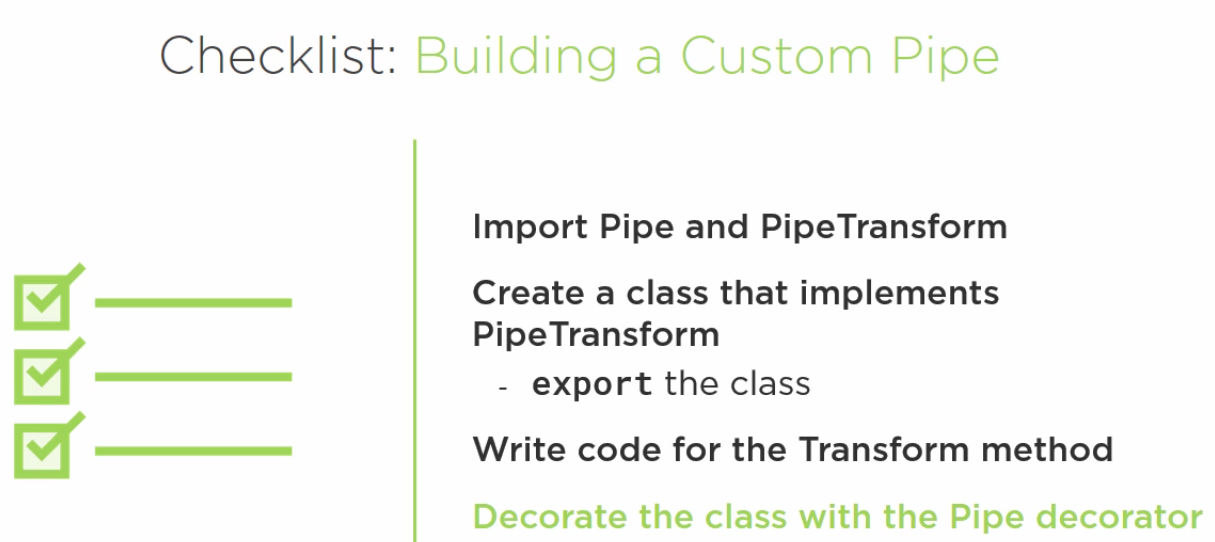
Procedimos a Crear un **Building Custom Pipes (ConverToSpacesPipe)**

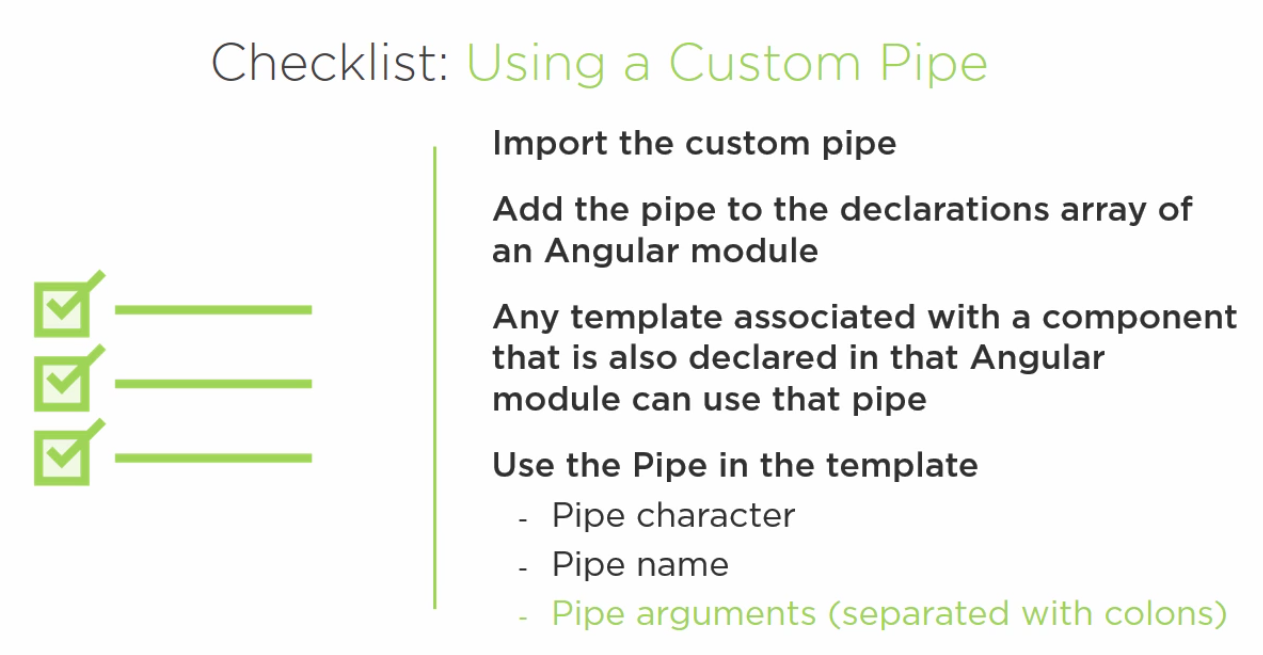


Luego de crear el archivo, lo agregamos en el archivo de HTML en la propiedad que queremos agregar el PIPE (Tuberia).



Pero para que todo funciones debemos de agregar o declararar el componente que creamos (ConverToSpacesPipe) al app.modules.ts para que pueda ser utilizado.



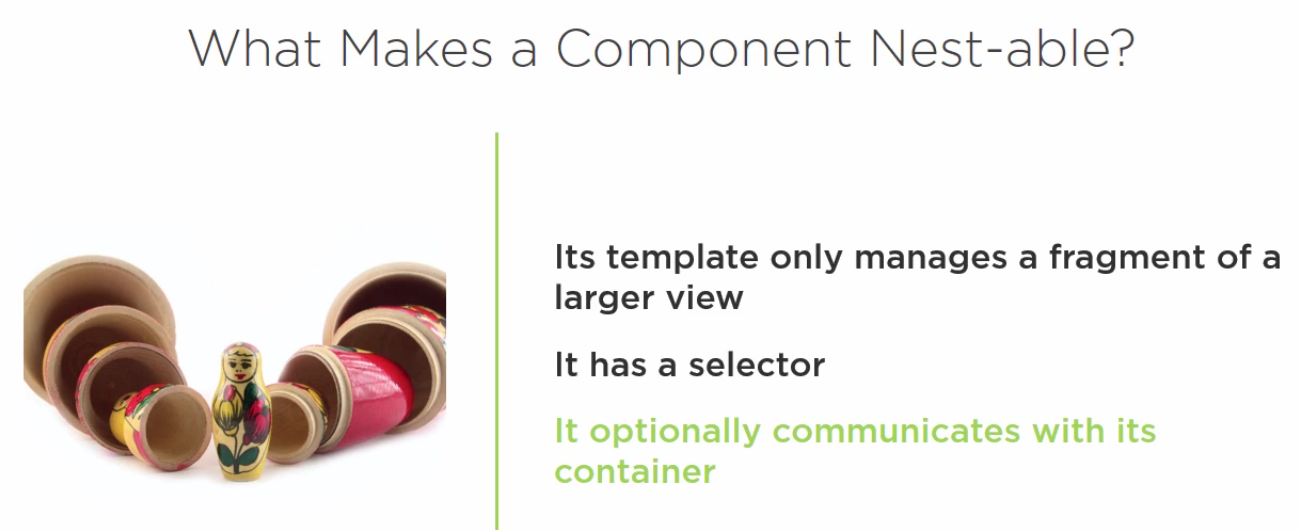


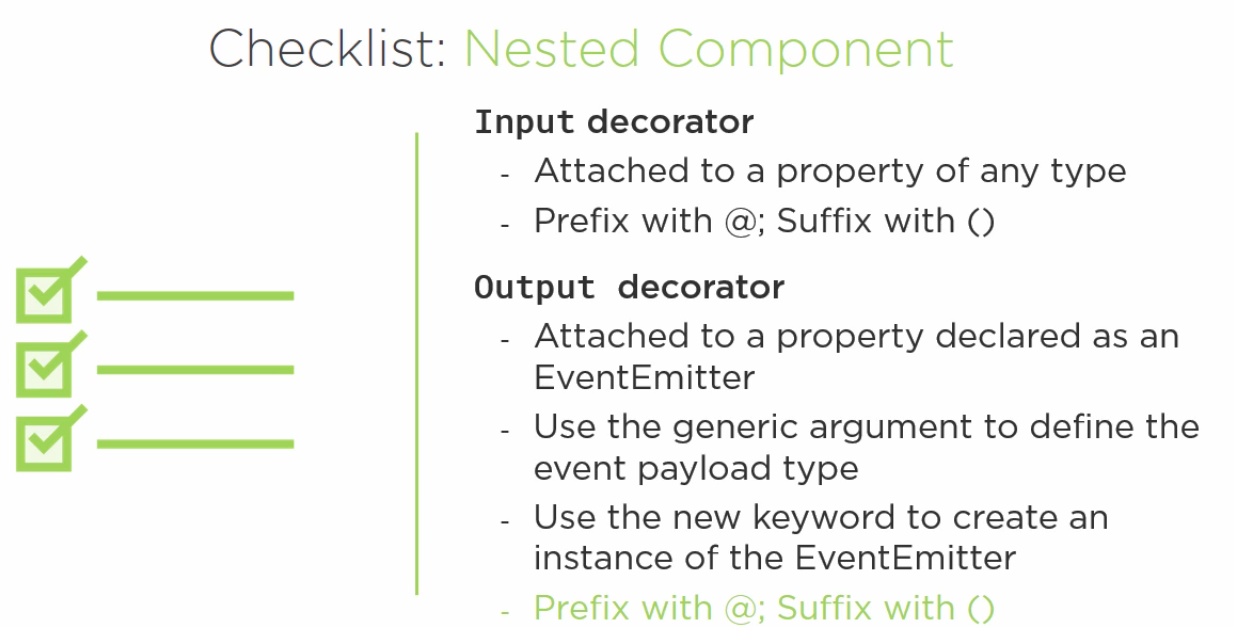
**Para Filtros**

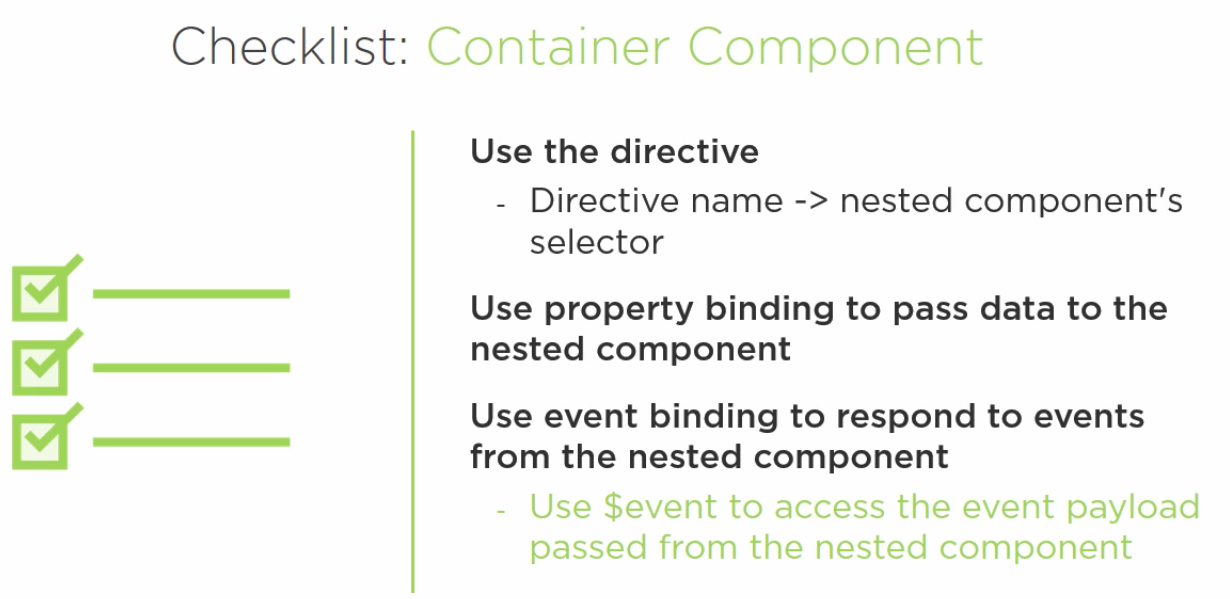
<https://developer.mozilla.org/en-US/docs/web/JavaScript/Reference/Global_Objects/Array/filter>

**8 – Building Nested Components**











**9 – Services and Dependency Injection**

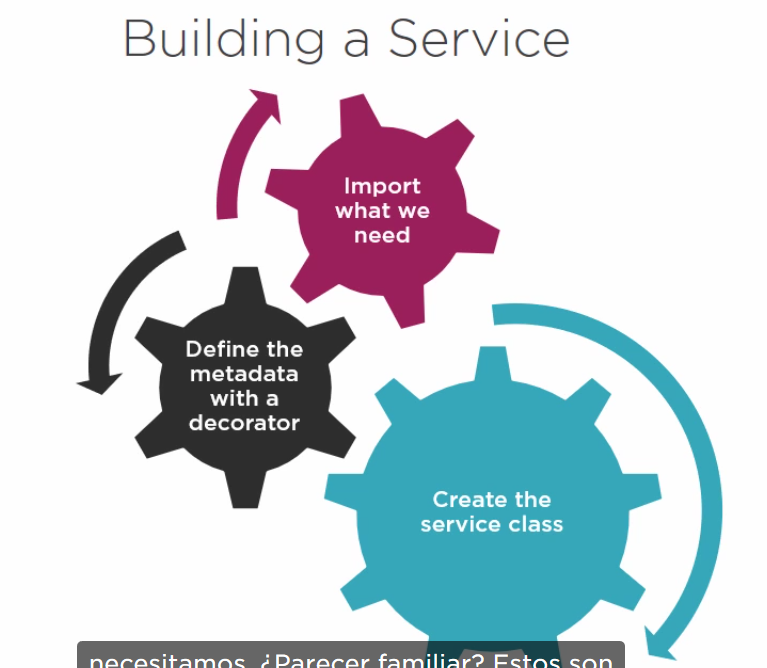
Un servicio es una clase con un proposito enfocado. A menudo creamos un servicio para implementar una funcionalidad que es independiente de cualquier componente en particular, para compartir datos o logica entre componentes o para encapsular interacciones externas como el acceso a Datos.

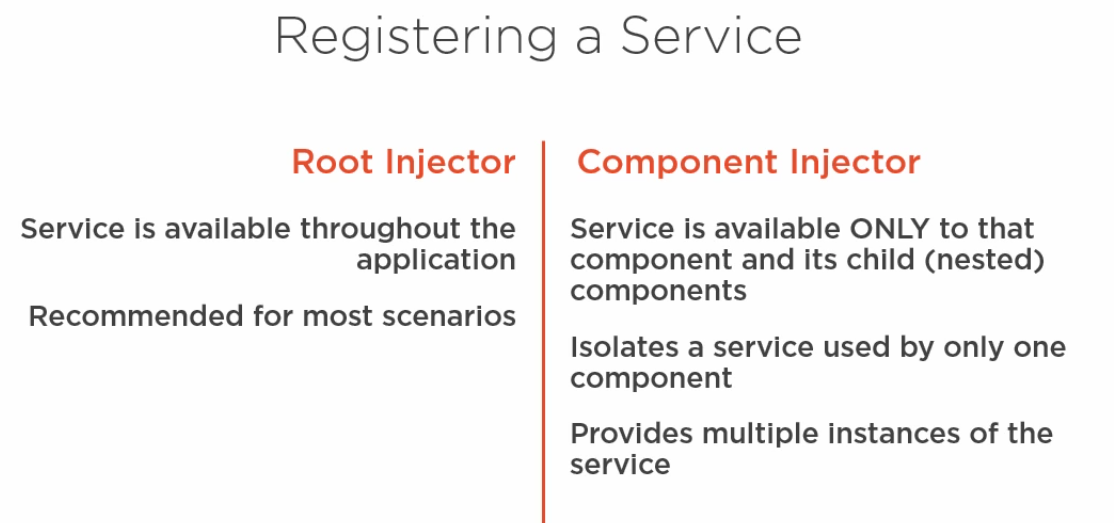


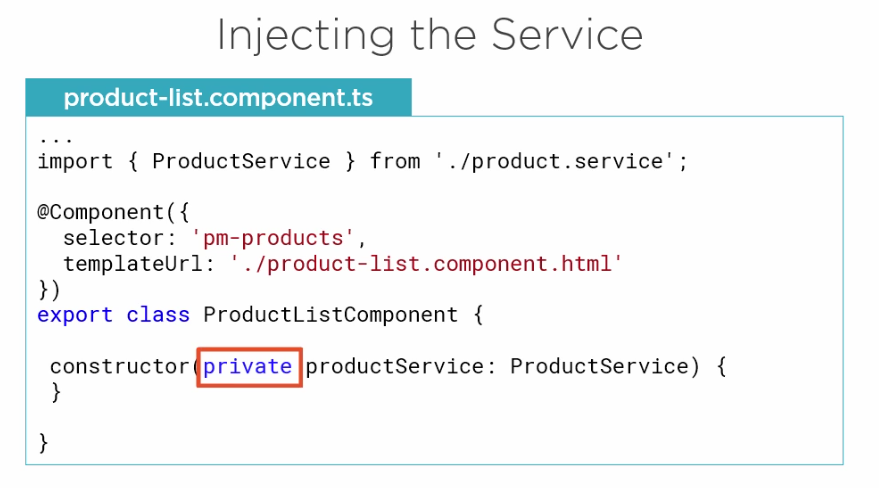
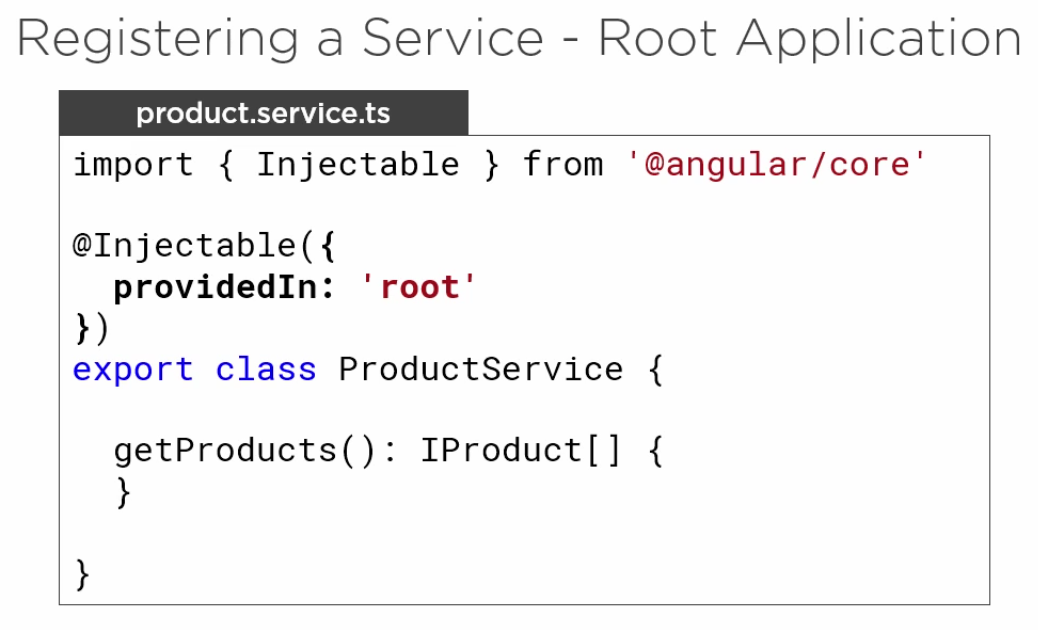
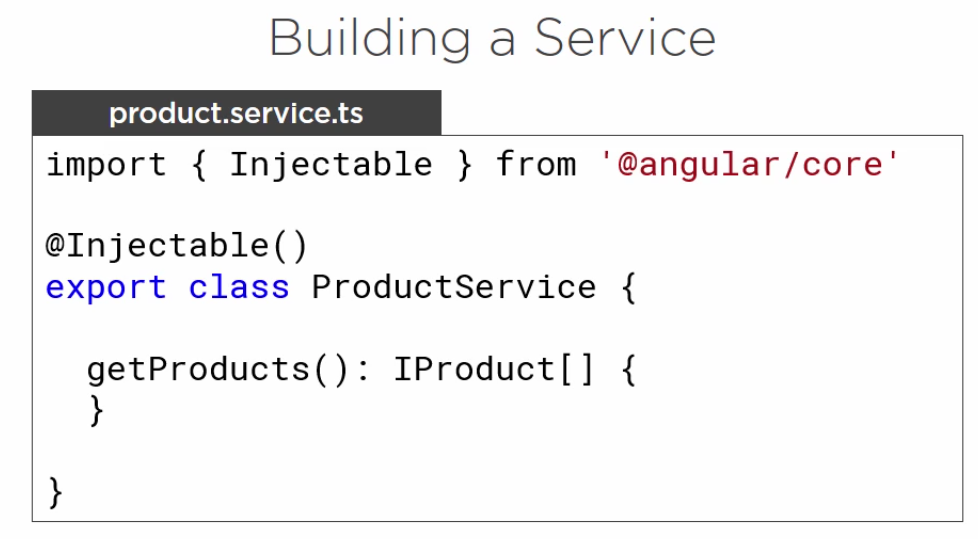
La inyeccion de Dependencia (Dependency Injection) es un patron de codificacion en el que una clase recibe las instancias de los objetos que necesita, denominados dependencias, desde una fuente externa en lugar de crearlos ellos mismos.

**Pasos para crear un servicio (Building a Service).**

1. Creamos la clase de servicio.
2. Definimos la metadata con un Decorador (Decorator)
3. Importamos lo que necesitamos







Concluciones:



