**Universidad de Alcalá**



**Escuela Politécnica Superior**

**Máster Universitario en Ingeniería del Software para la Web**

Trabajo Fin de Máster

Estudio sobre Angular 2 y superior

Piero Rospigliosi Beltrán

2020

Universidad de Alcalá

Escuela Politécnica Superior

máster universitario en

ingeniería del software para la web

Trabajo Fin de Máster

“Estudio sobre Angular 2 y superior”

**Autor: Piero Rospigliosi Beltrán**

**Director: Carlos Delgado Hita**

Tribunal:

Presidente: ................................................

Vocal 1º: .................................................... Vocal 2º: ....................................................

Calificación: ..................................................

Fecha: ..... de ................................ de .........

En agradecimiento a todos mis compañeros y profesores que siempre estuvieron dispuestos a prestar ayuda.

Índice Resumido

[1. Introducción 1](#_Toc177754577)

[2. Objetivos del Proyecto 5](#_Toc177754578)

[3. Estado del Arte 9](#_Toc177754579)

[4. Estandarización y Legislación en la Web 53](#_Toc177754580)

[5. Evaluación de Accesibilidad Web 129](#_Toc177754581)

[6. Resumen y Conclusión 263](#_Toc177754582)

[7. Bibliografía 273](#_Toc177754583)

[8. Apéndice A. Herramientas de Evaluación y Reparación de Accesibilidad Web i](#_Toc177754584)

[9. Apéndice B. Glosario xxxvii](#_Toc177754585)

[10. Apéndice C. Páginas con Certificados de Accesibilidad Web xlv](#_Toc177754586)

Índice Detallado

[1. Introducción 1](#_Toc177754587)

[2. Objetivos del Proyecto 5](#_Toc177754588)

[3. Estado del Arte 9](#_Toc177754589)

[3.1. Discapacidad 12](#_Toc177754590)

[3.1.1. Discapacidad visual 13](#_Toc177754591)

[3.1.1.1. Ceguera 13](#_Toc177754592)

[3.1.2. Discapacidad auditiva 20](#_Toc177754595)

[3.1.2.1. Sordera 20](#_Toc177754596)

[4. Estandarización y Legislación en la Web 53](#_Toc177754628)

[4.1. Estandarización 56](#_Toc177754629)

[4.1.1. Estándar de iure 57](#_Toc177754630)

[4.1.2. Estándar de facto 58](#_Toc177754632)

[4.1.2.1. AENOR. Asociación Española de Normalización y Certificación 59](#_Toc177754633)

[4.1.2.2. ANSI: Instituto Nacional Americano para Estándares 60](#_Toc177754634)

[5. Evaluación de Accesibilidad Web 129](#_Toc177754683)

[5.1. Herramientas de Evaluación de Accesibilidad Web 132](#_Toc177754684)

[6. Resumen y Conclusión 263](#_Toc177754745)

[6.1. Seguimiento del proyecto 266](#_Toc177754746)

[7. Bibliografía 273](#_Toc177754749)

[8. Apéndice A. Herramientas de Evaluación y Reparación de Accesibilidad Web i](#_Toc177754750)

[9. Apéndice B. Glosario xxxvii](#_Toc177754751)

[10. Apéndice C. Páginas con Certificados de Accesibilidad Web xlv](#_Toc177754752)

Índice de Figuras

**1. INTRODUCCIÓN**

**2. OBJETIVOS DEL PROYECTO**

**3. ESTADO DEL ARTE**

[Figura 1. Clasificación de las discapacidades 12](#_Toc177754914)

[Figura 2. Paletas de colores según deficiencias visuales 16](#_Toc177754915)

[Figura 3. Paleta sin adaptación de colores 17](#_Toc177754916)

[Figura 4. Paleta para alteraciones a colores Azul-Amarillo 18](#_Toc177754917)

**4. ESTANDARIZACIÓN Y LEGISLACIÓN EN LA WEB**

[Figura 22. Estados implicados en el proceso de desarrollo de un estándar ISO 65](#_Toc177754935)

[Figura 23. Estructura de la WAI 72](#_Toc177754936)

**5. EVALUACIÓN DE ACCESIBILIDAD WEB**

[Figura 40. Tipos de Herramientas 134](#_Toc177754953)

[Figura 41. Resultado de WebXACT en http://www.uah.es. Apartado General 143](#_Toc177754954)

**6. RESUMEN Y CONCLUSIÓN**

[Figura 155. Planificación Inicial del Proyecto 266](#_Toc177755068)

**7. BIBLIOGRAFÍA**

**8. APÉNDICE A. HERRAMIENTAS DE EVALUACIÓN Y REPARACIÓN DE ACCESIBILIDAD WEB**

[Figura 160. Herramientas de Evaluación y Reparación de Accesibilidad Web xxxv](#_Toc177755073)

**9. APÉNDICE B. GLOSARIO**

**10. APÉNDICE C. PÁGINAS CON CERTIFICADOS DE ACCESIBILIDAD WEB**

[Figura 161. Páginas con Certificados de Accesibilidad Web xlvii](#_Toc177755074)

Introducción

Los frameworks como Angular surgen de la evolución natural de la programación hacia formas que nos permitan programar a más alto nivel, dando al programador mayor flexibilidad para la abstracción y el centrarse en los detalles propios de cada aplicación a desarrollar.

Así mismo también, de la necesidad de crear aplicaciones cada vez más robustas, lo cual nos conduce a trabajar con frameworks y lenguajes de programación que nos brinden mayor capacidad para el mantenimiento del código y su reutilización.

Esta evolución y aumento de necesidades no es ajeno al desarrollo web en la parte del cliente, comúnmente denominado front-end. Y se debe a la propia naturaleza del desarrollo web la cual está basada en la arquitectura cliente/servidor. En la cual un usuario (cliente), a través de una interfaz realiza peticiones al servidor. El servidor, al recibir la petición, se encarga de procesar la información y devolver el recurso solicitado al cliente, quien se encargará de presentar la información obtenida en el formato adecuado. Debido a que el número de servidores es limitado, interesa trasladar cada vez más responsabilidades a la capa del cliente, ya que estos serán procesados en el ordenador del cliente aligerando la carga de trabajo del servidor.

El tipo de responsabilidades que se han ido trasladando al cliente tienen que ver desde, el renderizado de la información, validaciones en formularios, seguridad, etc.

En este sentido Angular y Typescript, lenguaje de programación utilizado por Angular ofrecen una serie de ventajas que se serán explicadas en detalle en los siguientes capítulos de este trabajo.

Por su parte Typescript, que es una extensión de Javascript, tiene como particularidad el tipado, la orientación a objetos y su alineamiento con estándares nuevos de Javascript como EcmaScript 6 (ES6).

Angular, nos ofrece controladores, servicios, directivas para organizar el proyecto y características tan importantes como el two-way data binding que nos permite reflejar automáticamente cambios en nuestro modelo en el DOM de nuestra aplicación, tarea que tradicionalmente se hacia de forma manual.

.

Objetivos del Proyecto

Uno de los objetivos de este documento es hacer un repaso de la historia y características de este framework y de los componentes que lo que integran, partiendo desde los más básicos como Javascript, Typescript, EcmaScript y la relación entre ellos.

Mediante el repaso detallado de sus características se pretende explicar como cada una de ellas nos facilita el desarrollo de código limpio, robusto y mantenible.

También explicaremos la evolución de Angular, y la aparición de frameworks similares con la misma forma de trabajar.

JAVASCRIPT

Es un lenguaje de programación que permite crear acciones en las páginas web. Es un lenguaje de programación nativo de los navegadores quienes son los encargados de interpretar estos códigos.

Se define como orientado a objetos, basado en prototipos, débilmente tipado, dinámico, imperativo e interpretado.

ECMASCRIPT

Es el estándar que define como debe ser el lenguaje Javascript. Dicho de otra manera es la especificación que deben seguir los creadores de software para crear intérpretes para Javascript.

Javascript siendo un lenguaje interpretado que ejecuta las instrucciones directamente sin una previa compilación , cuenta con multitud de intérpretes como como los propios navegadores web o servidores como NodeJS.

Para establecer las reglas de juego que deben cumplir dichos intérpretes, ECMA Intenational publica el estándar ECMAScript.

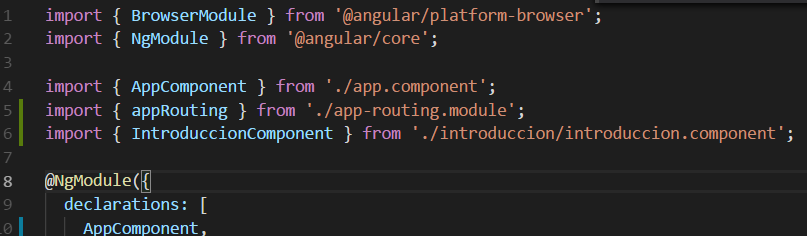
ECMAScript ha ido evolucionado con el tiempo y es en el año 2015 donde se lanza ECMAScript 6, siendo el estándar que utiliza TypeScript que se explicará en el siguiente apartado.

ECMAScript 6 introduce cambios importantes como:

-El uso de clases, que nos permite utilizar estructuras predefinidas como molde para crear objetos. Estos nos permiten declarar atributos y métodos, que en Javascript clásico equivaldrían a una variable a una función. La aparición de las clases nos brinda un valor importantísimo porque nos permite reutilizar con mayor facilidad nuestras estructura de datos.

IMAGEN CLASES

-El concepto de módulos que nos permite definir archivos que contienen directivas de importación y exportación a nivel superior.



-Declaraciones de variables, const y let. La primera nos permite crear constantes, y la segunda variables locales en nuestros bloques. Esto también nos facilita crear aplicaciones más robustas, ya que podemos usar estas declaraciones para asegurar el alcance de estas y si se puede modificar o no.

IMAGEN const let

-Otras características que nos permiten diseñar mejor nuestra lógica de negocio como el poder tener valores por defecto en nuestras funciones, el uso de lambdas o poder utilizar el paradigma de la programación funcional. Un ejemplo de estos son las funciones de orden superior, map, filter y reduce

.map: El resultado nos devuelve una colección de elementos que son el resultado de la expresión aplicada al atributo o elemento creado.

Por ejemplo: [1,2,3].map(n => n+1) . En este ejemplo se tomaría cada uno de los valores, que vendrían a ser la n y se sumaría 1, de esta forma el resultado esperado sería [2,3,4]

. filter: El resultado tendrá la misma forma que el argumento de entada, sin embargo solo se quedará con los elementos que cumpa la condición dada.

Por ejemplo: [1,2,3,4].filter(n=> n < 3) . Este caso le estamos diciendo que solo se quede con los valores menores que 3, por lo que el resultado esperado sería [1,2]

.reduce: Nos permite aplicar una función contra un acumular y cada valor de la matriz (de izquierda a derecha) para reducirlo a un solo valor.

Por ejemplo [1,2,3,4].reduce((acumulador, valorActual) => acumulador + valorActual). En este caso cada valor se sumará al actual valor acumulado hasta dicho momento. Actuando en este ejemplo como una sumataria donde el resultado sería 10.

Siendo los pasos:

0+1 => 1

1+2 => 3

3+3=> 6

6+4=> 10

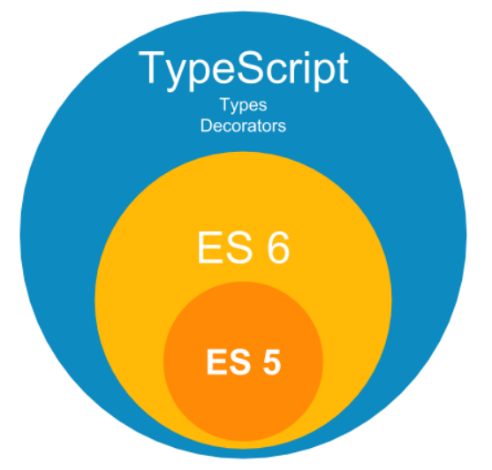
Como vemos cada nuevo valor se suma al anterior valor acumulado en dicho momento.

Este tipo de funciones nos permiten abstraernos de cómo se hace, y centrarnos en lo que queremos hacer para definir nuestra lógica de negocio de manera más clara y a su vez crear funciones más reutilizables y mantenibles.

IMAGEN EJEMPLO PROGRAMACION FUNCIONAL

## Typescript

De ECMAScript6 surge Typescript, que es un superset del primero. Esto quiere decir que incluye todas las funcionalidades de ES6 pero añade funcionalidades extra.



Entonces, ¿Qué funcionalidades extras ofrece con respecto ES6, para que la comunidad de Angular prefería utilizarlo en lugar del estándar?

Las características extras mas importantes con respecto a ES6, son:

-Es un lenguaje compilado: Lo cual es una ventaja porque nos permite encontrar errores en tiempo de compilación antes de abordar la ejecución. El compilador de Typescript transforma el código a código Javascript tradicional el cual puede interpretado por todos los navegadores.

-Variables tipadas: Nos permite a los programadores utilizar herramientas y prácticas de desarrollo altamente productivas: verificación estática, refactorización de código, finalización de declaraciones y navegación basada en símbolos.

Por ejemplo:

Definimos un array de tipo number

numeros:Array<number>

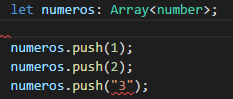
Luego añadimos dos elementos:

numeros.push(1);

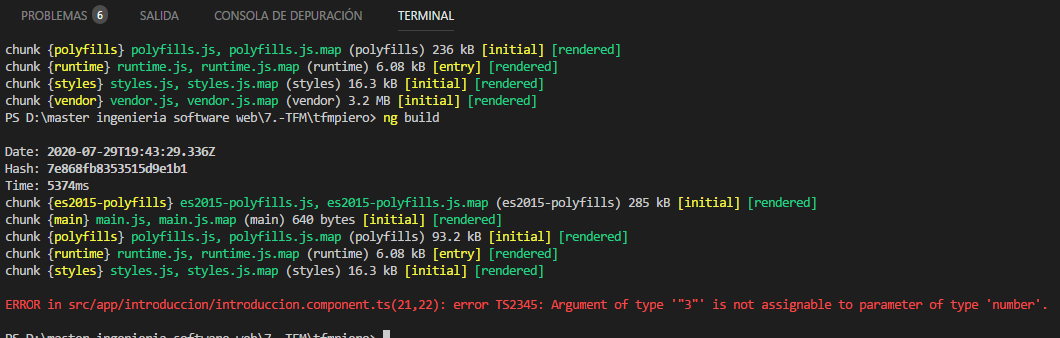
numeros.push(2);

Hasta el momento tenemos al array [1,2]. Ahora intentamos añadir un número, pero en forma de string.

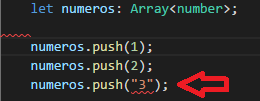
numeros.push(“3”);



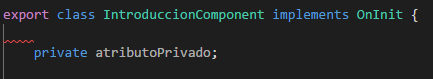
Esto en javascript clásico, ES5, no daría ningún problema. Sin embargo en Typescript veríamos un error en tiempo de compilación.



Esto nos ofrece una gran ventaja, porque nos evitamos encontrar el error en tiempo de ejecución. Incluso si utilizamos un editor adecuado, este mismo nos señalará el error al instante, sin necesidad de compilar el código.

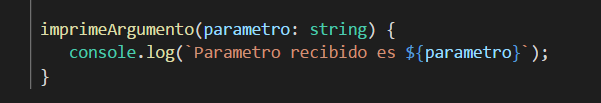


-Modificadores de acceso público, private y protected: Los cuales nos permiten definir el alcance de un miembro de una clase. Nuevamente, esto nos permite detectar errores en tiempos de compilación haciendo que nuestra aplicación sea más robusta.

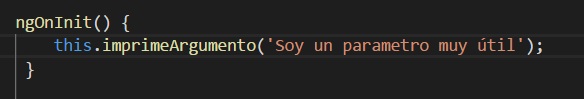


-Decoradores: Nos permite implementar el patrón decorador, el cual nos permite añadir funcionalidad extra de forma dinámica, lo cual nos permite evitar tener que crear nuevas clases que heredan de nuestra clase a expandi..

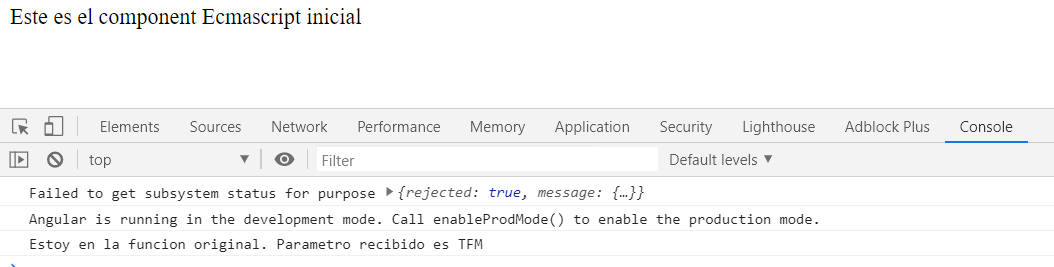
Por ejemplo imaginemos que tenemos la clase llamada ECMAScript la cual tiene una función que nos imprime el argumento pasado por parámetro.



Llamamos a la función



El resultado de esta función tal como se espera es la impresión en consola del parámetro recibido.



Sin embargo, ahora se requiere que cuando se llame a ciertas funciones que consideramos importantes, se imprima información relativa a su ejecución, como la clase de donde se invoca y los argumentos recibidos.

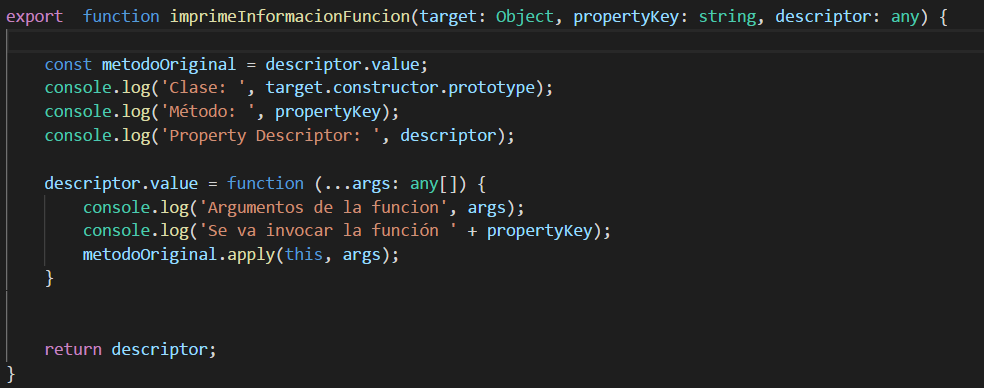
Podríamos modificarlas funciones deseadas y añadir esta funcionalidad o recurrir a la herencia y sobrescribir el método imprimeArgumento mediante nueva una nueva clase. Sin embargo, el principio de responsabilidad única (Single Responsability Principle, SRP) de SOLID numerito y explicación de solidº , nos indica que una clase o módulo debe tener uno y una solo responsabilidad.

Si nuestra función tiene un objetivo, que es imprimir argumentos, cualquier otra acción secundaria nos dificultará la legibilidad, mantenimiento y la escalabidad de nuestra función y nuestra aplicación.

Por ello vamos aprovechar la ventaja que nos ofrecen los decoradores y esconder todo el código secundario nuevo para que no ensucie nuestra función. De esta forma, el siguiente desarrollador a cambiar la funcionalidad principal, solo tendrá que preocuparse de eso, olvidándose de las acciones secundarias teniendo así nuestro código mayor legibilidad.

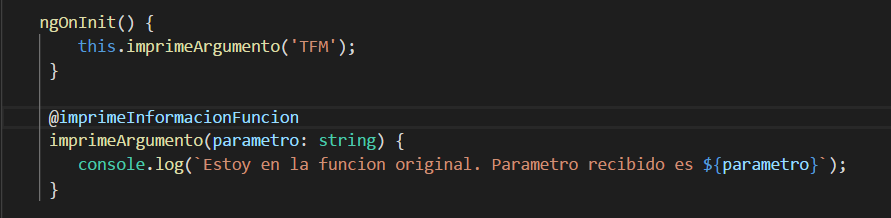
La manera en que funciona el decorador es que primero se entrará por la función del decorador y podemos realizar acciones con los metadatos recibidos de la función decorada.

Se crea del decorador el cual como podemos observar tiene una forma predefinida en los argumentos que nos facilita Typescript.

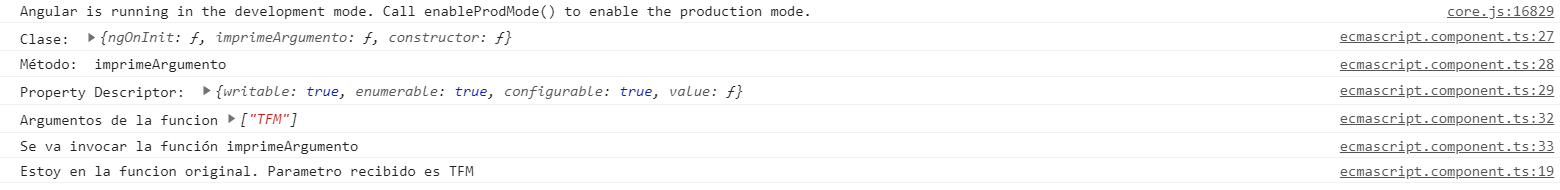


Vemos que imprimimos en consola la información de los metadatos de la función. Y luego invocamos a la función original para que se continue haciendo la funcionalidad principal.

Ahora podemos decorar nuestro método con nuestro nuevo decorador.



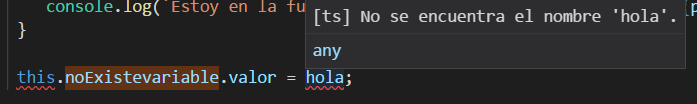
Vemos el resultado en la consola, y efectivamente ha impreso los metadatos para luego invocar la función original.



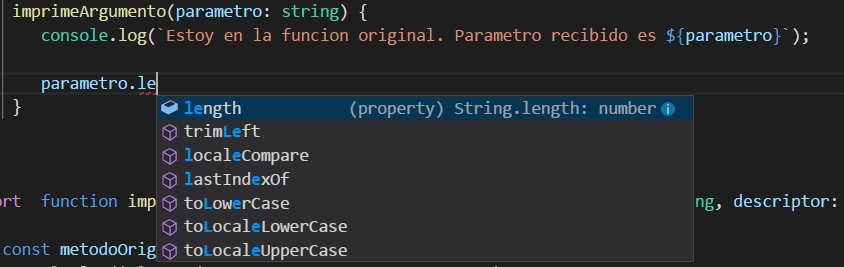
De esta forma hemos añadido código nuevo y secundario de forma elegante y sin ensuciar nuestra funcionalidad principal.

-Editores para trabajar con TypeScript. Es una experiencia más agradable y eficiente trabajar con un editor como Visual Studio Code, que nos permite tener autocompletado o detección errores al momento de programar.

Por ejemplo aquí vemos como nos indica que no existe la variable hola.



Y aquí nos sugiere mediante un autocompletar las posibles funciones a utilizar para la variable string parámetro



-El apoyo de grandes empresas y proyectos. Empresas como Microsoft, el creador de TypesScript. O Google, quien mantiene el framework Angular, el cual es objeto de estudio en este trabajo.

### Angular

Es un framework para crear aplicaciones cliente una sola página (SPA) eficientes y sofisticadas utilizando HTML y Typescript.

Desarrollado en Typescript, por Google, nos permite disponer de herramientas para que el desarrollo y las pruebas sean más fáciles.

Angular es la evolución de AngularJS, aunque incompatible con su predecesor. Esto se debe a que es un framework totalmente nuevo que surge como solución a las carencias que mostraba AngularJS con respecto a problemas de rendimiento, escalabilidad, modularidad y también el soporte en móviles.

#### Arquitectura

La arquitectura de una aplicación Angular se basa en dos componentes fundamentales los cuales son los módulos y los componentes. Los bloques de construcción básicos son los módulos o también llamados NgModules, los cuales contienen a los componentes, quienes junto a sus plantillas definen una vista.

Los módulos proporcionan un contexto de compilación para los componentes y permiten descomponer nuestras funcionalidades en bloques individuales que exponen interfaces de comunicación bien definidas. Una aplicación Angular está definida por un conjunto de módulos y siempre tiene al menos un módulo raíz que permite el arranque y carga todos los demás módulos.

En la ingeniería del software este tipo de arquitectura se denomina arquitectura basada en componentes, el cual es una enfoque que se centra en al descomposición del diseño en componentes funcionales o lógicos.

Una aplicación angular se basa en la unión de distintas piezas de código los cuales son diseñados para cumplir con un propósito particular y que agrupan todo lo relacionado con alguna funcionalidad concreta de forma aislada haciendo explícito una interfaz para ofrecer sus servicios. Y dado que los módulos son quienes representan estas piezas de código, éstos vendrían a representar el papel de componentes desde el punto de vista de una arquitectura basada en componentes.



Los principios fundamentales en los que se basa la arquitectura basada en componentes son:

-Reusabilidad: Los componentes son diseñados para poder ser utilizados en distintos escenarios, ya que agrupan todo lo necesario para poder desempeñar su tarea específica.

-Libre de contexto: Nos indica que debe poder operar en diferentes ambientes y contextos por lo que información espeficifica como el estado de los datos deben ser pasados al componente y no estar incluidos en ellos.

- Extensibles: Puede expandir su funcionalidad o crear una nueva.

-Encapsulado: Exponen su funcionalidad mediante interfaces, evitando revelar detalles internos del componente.

- Independientes: Los componentes deben ser diseñados para no tener dependencia de otros componentes. Por lo que deben poder ser instalados sin afectar a otros componentes o s

De esto se desprende que Angular mediante el uso de esta arquitectura intenta hacer suyo los principios mencionados permitiéndonos la creación de código más limpio, robusto y mantenible.

La documentación oficial de Angular nos incluye una gráfica donde se nos explica la relación de los elementos que componen su arquitectura y que serán explicados con más detalles en los siguientes apartados.

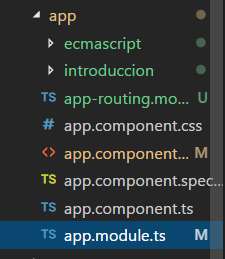


En Angular los módulos, componentes y servicios son clases que usan decoradores. Estos decoradores marcan su tipo y proporcionan metadatos que le dicen a Angular cómo usarlos.

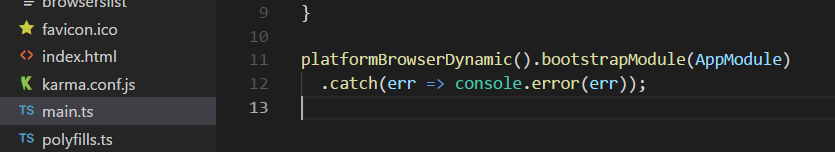
Módulos

Como se mencionó anteriormente los módulos o NgModules vendrían a representar esas piezas de código reutilizables que agrupan a modo de contenedor un bloque cohesivo de código dedicado a un dominio de aplicación, un flujo de trabajo o un conjunto de capacidades estrechamente relacionadas que suelen estar estrechamente ligadas al desarrollo de una funcionalidad.

Pueden contener componentes, proveedores de servicios y otros archivos de código cuyo alcance está definido por el NgModule que los contiene. Un módulo puede importar a otro módulo y con ello su funcionalidad. Un módulo a su vez puede exportar su funcionalidad para que la utilicen otros módulos.



Cada aplicación Angular tiene al menos una clase módulo, el módulo raíz, el cual convencionalmente se denomina AppModule y reside en el archivo app.module.ts. Este módulo será cargado desde un archivo principal, normalmente main.ts, el cual será el punto de entrada a la aplicación.

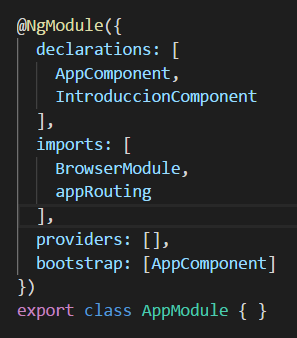


El módulo raíz puede contener todos los módulos secundarios que necesite, los cuales a su vez pueden contener otros módulos representando una jerarquía de cualquier profundidad.

Composición de un Modulo

El decorador @NgModule

Un módulo se define mediante una clase decorada con @NgModule. Este decorador es una función que que indica al framework que se trata de un módulo y cuales son las propiedades que lo describen. Estas propiedades serán descritas en los posteriores apartados.



El array declarations:

Se trata de los componentes, directivas, y pipes que pertenecen a este módulo.

El array exports:

Es el subconjunto del array declarations que deseamos que sean visibles y utilizables por otros módulos que importen este módulo.

El array providers:

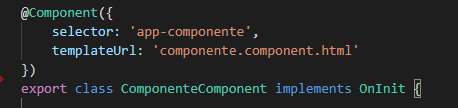
Aquí declaramos todos aquellos servicios que pasan a ser parte de la colección global de servicios. Es decir, se vuelven accesibles desde cualquier parte de la aplicación.

El array Bootstrap:

Esta propiedad solo debe ser llamada por el módulo raíz. Aquí se llamada al componente principal el cual aloja a todas las demás vistas de la aplicación.

Componentes

Un componente controla una zona de espacio de la pantalla a través de las vistas. Un componente es el elemento que define la vista. Un componente es una clase estándar decorada con @Component.



Dicha clase define la lógica de la vista y tiene asociado un plantilla que contiene el código HTML junto a directivas de Angular que afectan su comportamiento.

COMPONENTES Y DATA BINDING

Los componentes son dinámicos. Esto quiere que Angular los va creando, actualizando o destruyendo conforme el usuario se mueve por la aplicación. Al encargarse Angular de la actualización del DOM nos quita esta responsabilidad a los desarrolladores y nos permite crear un código más limpio enfocándose solo en la lógica de negocio.

Sin Angular, los programadores serían los responsables de introducir nuevos valores de datos en el código HTML y de convertir las respuestas de los usuarios en acciones y actualizaciones de valores. Dejar esto en mano de los programadores puede resultar tedioso, propenso a error, y más difícil de leer y por tanto de mantener. Data Binding nos permite delegar esta responsabilidad al framework.

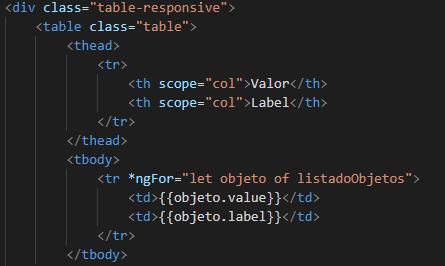
Data Binding nos permite mostrar contenido dinámico en lugar de estático. El Data Binding es la comunicación entre nuestro código HTML y nuestra lógica de programación ubicado en nuestro archivo .ts. Y lo que logra es cualquier cualquier en el modelo de datos de nuestra clase se refleje inmediatamente en nuestra plantilla HTML.

Este es un enlace de datos bidireccional, un mecanismo para coordinar las partes de una plantilla con las partes de un componente. Lo cual quiere decir que el DOM puede modificar la clase y la clase puede modificar al DOM.

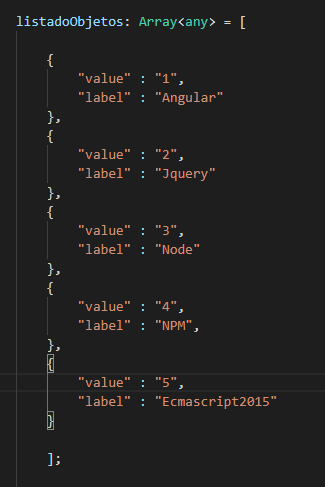


Por ejemplo:

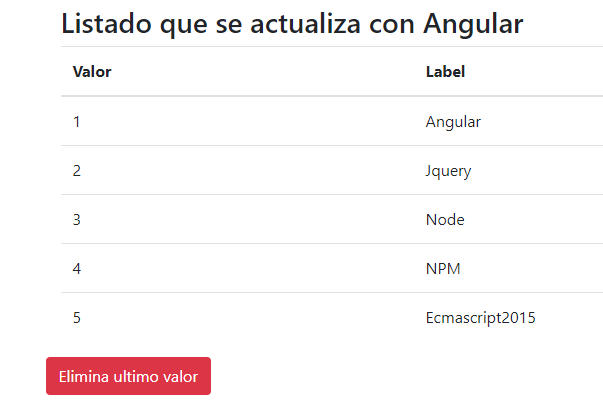
En Angular, para construir los elementos de una tabla nos basta con recorrer los elementos de un array.



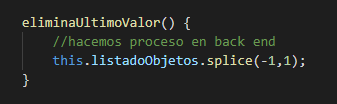
Siendo la variable “listadoObjetos” un array de objetos con 5 valores que se pueden ver a continuación



Angular nos construye los tr de manera dinámica gracias a la directiva \*ngFor:



Si quisiéramos actualizar los valores de dicha tabla, nos bastaría con actualizar la variable “listadoObjetos”. Esto es importante porque con otra tecnología como Jquery tendríamos que manualmente eliminar, actualizar o agregar un tr. Lo cual nos permite centrarnos en nuestra lógica de negocio, la cual consiste en actualizar nuestra variable que representa a nuestros datos.



SERVICIOS

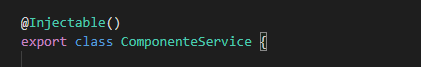
Los servicios son las clases que abarcan cualquier valor, función o característica que necesita una aplicación. Tiene típicamente un propósito limitado y bien definido y aportan a los desarrolladores la capacidad de reutilización del código y mantener los componentes lo más desacoplados posible.

Esto gracias a que los servicios en Angular trabajan mediante el patrón de diseño inyección de dependencias. El funcionamiento consiste en que Angular instanciará nuestros servicios y los dejará disponibles para que un módulo o componente los inyecte y estén disponibles para todo el elemento.

Es decir, un mismo servicio puede ser inyectado en un módulo y estar disponible en todo el módulo, sin necesidad de importarlo en todas las clases que se vaya a utilizar y sin necesidad de instanciar dicha clase pues Angular lo habrá hecho por nosotros.

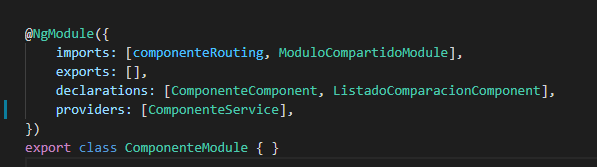
La función principal de un servicio en Angular es interactuar con el backend de la aplicación en busca de los datos. Un componente interactúa con el backend a través de un servicio. Este servicio además puede ser reutilizado en todos los sitios donde se inyecte.

Un servicio es una clase normal el cual lleva el decorador @Injectable, el cual proporciona los metadatos necesarios para que Angular pueda inyectar dicha clase.



SERVICIOS EN MODULOS

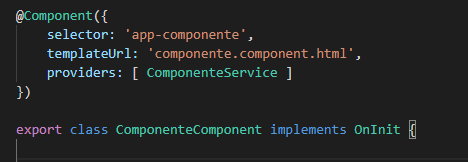
Como se me mencionó un servicio puede estar inyectado en un módulo. De manera que todos los elementos del módulo tengan acceso al servicio. De esta forma todos los componentes de un módulo concreto pueden acceder al servicio que se ha registrado en el array providers correspondiente.



SERVICIOS EN COMPONENTES

Un servicio puede ser declarado directamente en un componente concreto. Lo cual significará que dicho servicio solo estará disponible para ese componente en concreto, o por sus componentes hijos.

Cuando se inyecta un servicio directamente en un componente, este servicio se creará solo en el momento que se cree el componente y su ciclo de vida dependerá del ciclo de vida del componente.

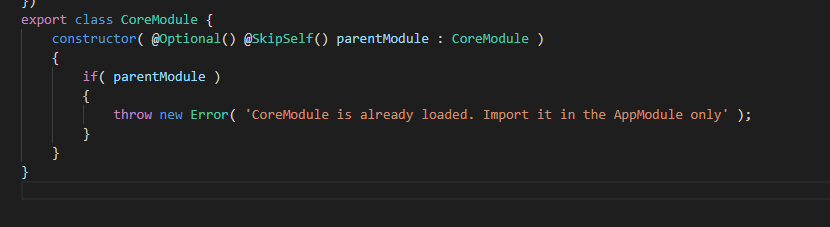


SERVICIOS Y EL PATRON SINGLETON

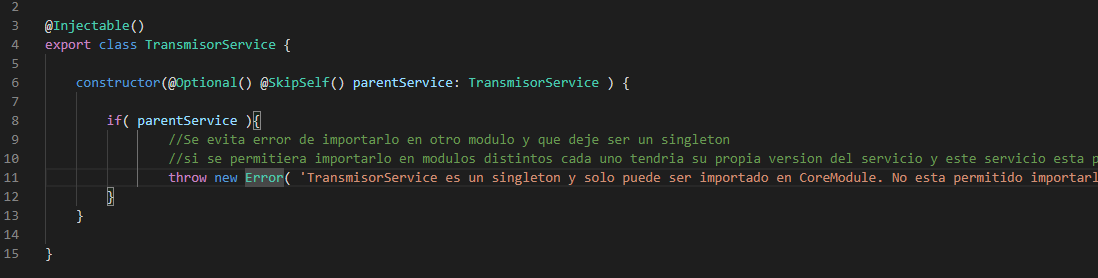
El patrón de diseño Singleton consiste en restringir la creación de un objeto a una única instancia. Angular nos permite aprovechar la ventaja de este enfoque y utilizarlo para nuestros servicios. Es decir, podemos hacer que un servicio determinado sea un Singleton.

La ventaja de que un servicio sea un Singleton viene del hecho que podemos asegurar que los datos se manipulen en él no estarán sufriendo cambios no deseados desde algún otro componente. Por ejemplo, si se está transfiriendo datos desde un componente hacia otro componente a través de un servicio, es de vital importancia que ninguno de los dos componentes implicados pueda instanciar nuevamente al servicio que sirve como conductor.

La forma más común de hacer esto es crear un módulo llamado CoreModule el cual se inyectará en el módulo raíz y quién contendrá nuestros servicios singleton. Además, CoreModule utilizará las anotaciones @Optional y @SkipSelf para comprobar que no se instancie en ningún otro lado.



Una forma mas concreta de hacerlo es comprobando directamente en el servicio que no se instancie más de una vez, mediante el uso de las anotaciones antes mencionadas en su constructor.



DIRECTIVAS

Las directivas son una de las piezas clave que nos brinda Angular para manipular el DOM de manera más sencilla y por tanto tener código más mantenible.

Se puede decir que las directivas son pequeñas instrucciones de código que se aplican sobre el DOM. En cierto sentido, los componentes son un tipo de directiva, ya que cuando creamos un componente lo que estamos diciendo es que se construya el HTML a partir de nuestra clase .ts.

DIRECTIVAS ESTRUCTURALES

Las directivas estructurales son aquellas que se cambian el diseño del DOM, permitiendo añadir o eliminar sus elementos. Estas directivas son fácilmente reconocibles debido que están antecedidas por un asterisco (\*) seguido del nombre de la directiva.

En Angular tenemos directivas estructurales predefinidas y directivas estructurales personalizadas.

DIRECTIVAS ESTRUCTURALES PREDEFINIDAS

Aquellas que nos proporciona angular. Para poder usarlas hace falta importar el módulo CommonModule de @angular/core en el módulo deseado. Una buena práctica es importarlo dentro de un módulo compartido el cual contendrá este tipo de módulos predefinidas.





DIRECTIVA ESTRUCTURAL \*NFOR

Una directa de predefinida es la directiva \*nFor el cual nos permite crear elementos del DOM en bucle recorriendo un array.

Su sintaxis es \*ngFor=“let v of valores”, donde valores es el array a recorrer y v es la variable local donde se almacena el valor iterado,



En el ejemplo, se entiende que se crearán tantos tr como valores tenga el array listadoObjeto



La importancia de esta directiva reside en que si queremos modificar los elementos del DOM nuevamente, basta con cambiar la variable asociada. Angular se encargará del resto.

DIRECTIVA ESTRUCTURAL \*NGIF

La directiva \*ngIf nos permite crear o eliminar un elemento del DOM según el valor de la variable comprobada. Esta directiva no solo esconde o muestra el elemento, si no que lo construye o destruye lo que nos brinda un mayor rendimiento en el cliente.

Su sintaxis es \*ngIf=“condicion” donde “condicion” es el la expresión a comprobar y obtener resultado true o false. Esta directiva se aplica a todo el elemento y a sus hijos.

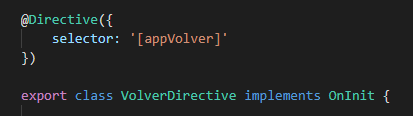


De la imagen anterior se puede deduce que el botón solo se mostrará si la expresión “listadoIObjectos.length” resulta true.

DIRECTIVAS ESTRUCTURALES PERSONALIZADAS

Angular nos da la posibilidad de crear nuestras propias directivas estructurales de manipulación del DOM. De forma que se pueda crear un comportamiento personalizado al elemento del DOM al que se le aplique.

Para ello debemos crear una clase decorada con @Directive donde el parámetro selector será el nombre a utilizar para identificar a la directiva cuando se aplique en las plantillas.



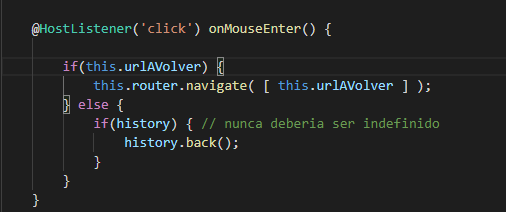
También es obligatorio importar las clases Input, TempleteRef y ViewContainer de @angular/core



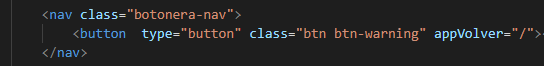
Además, para poder manipular desde la clase el elemento DOM desde la clase hace falta declarar en el constructor a ElementRef.



Para asociar eventos a una función que realizará las acciones deseadas se utiliza el decorador @HostListener, a quién se le pasa por parámetro el evento asociado



Para este ejemplo podemos observar que los elementos que tengan asociado la directiva appVolver, realizarán las acciones de la función onMouseEnter para el evento click.



DIRECTIVAS DE ATRIBUTOS

Son aquellas directivas que permiten cambiar la apariencia o el comportamiento de un elemento del DOM a diferencia de las directivas estructurales que permiten agregar o eliminar elementos del DOM.

Las directivas de atributo se utilizan como atributos de elementos.

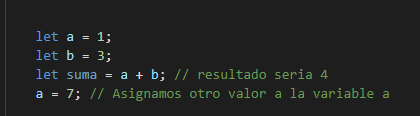
Un ejemplo es la directiva NgStyle, la cual es una directiva de atributo predefinida que nos permite cambiar varios estilos del elemento al mismo tiempo.

ANGULAR Y LA PROGRAMACION REACTIVA

La programación reactiva es un paradigma de programación asincróna que se ocupa de los flujos de datos (streams) y la propagación del cambio. Una manera sencilla de explicar este concepto sería comparándolo con la programación tradicional secuencial en Javascript.

Cuando programamos de forma tradicional las instrucciones se ejecutan una detrás de otra y el resultado no se altera si una variable se ha cambiado de forma posterior.

En el siguiente ejemplo, que se cambie el valor de la variable a no afectará a la variable suma, pues se ha cambiado después de que se ha realizado el calculo.



Sin embargo, mediante la programación reactiva la variable suma estaría constantemente atento a los valores de a y b, y si se cambian también cambiará su resultado.

En programación reactiva se pueden crear flujos de datos (streams) asíncronos de cualquier cosa, como por ejemplo los valores que toma una variable a lo largo del tiempo o los clicks sobre un botón.

La ventaja de esto es que nos ofrece sistemas que son capaces de consumirlos de distintos modos, pero centrándose en principalmente en los siguientes tipo de eventos:

-La respuesta o aparición de un evento dentro del stream

- La aparición de un error en el stream.

-La finalización del stream.

OBSERVABLES

Los observables son una implementación de la programación reactiva. Nos permite producir eventos y consumirlos de diversos modos.

Se basa en el patrón de diseño software “Observer” donde un sujeto mantiene una lista de dependientes, llamados observadores, y les notifica automáticamente los cambios de estado.

Los observables son declarativos, porque no se ejecutan hasta que un consumidor se suscribe. A su vez, el consumidor recibe notificaciones hasta que se completa la función o hasta que se da de baja. Los observables brindan soporte para pasar mensajes entre las partes de la aplicación y nos permiten el manejo de eventos, programación asíncrona y manejo de valores múltiples.

Un observable puede entregar múltiples valores de cualquier tipo: literales, mensajes o eventos, según el contexto. Con los observables el código desarrollado solo necesita preocuparse de suscribirse para consumir valores y cuando termine, si fuese el caso, cancelar la suscripción.

RxJS

Es una librería Javascript que contiene la implementación de los observables y que utiliza Angular. Esta librería será necesaria hasta que se parte el estándar JavaScript y los navegadores lo admitan de forma nativa.

OBSERVABLES EN ANGULAR CON RxJS

Como se mencionó anteriormente Angular hace uso de los observables de la librería RxJS para una variedad de operaciones asíncronas comunes.

Por ejemplo:

-Datos de salida de un componente secundario a un componente principal.

- En el módulo HTTP para manejar solicitudes y respuestas AJAX.

- En los módulos Router y Forms para escuchar y responder a los eventos de entrada del usuario.

TRANSMITIENDO DATOS ENTRE COMPONENTES

A través de la clase EventEmitter que se usa al publicar valores de un componente a través del decorador @Output. EventEmitter extiende a RxJS Subject y proporciona un método emit() para que puedan enviar valores mediante el metodo next() a cualquier observador suscrito.

OBSERVABLES EN EL MÓDULO HTTP.

En Angular HttpClient devuelve observables de llamadas al método HTTP. Esto ofrece varias ventajas con respecto a peticiones Http basadas en promesas:

-Los observables, no cambian la respuesta del servidor. En su lugar, puede utilizar una serie de operadores para transformar valores según sea necesario.

-Las solicitudes HTTP se pueden cancelar mediante el método unsubscribe()

-Las solicitudes fallidas se puede reintentar fácilmente.

ASYNCPIPE

Tubería asíncrona o AsyncPipe, se suscribe a un observable o promesa y devuelve el último valor que ha emitido. Cuando se emite un nuevo valor, la tubería marca el componente a verificar para ver si hay cambios. Cuando el componente se destruye, la canalización asíncrona cancela la suscripción automáticamente para evitar pérdidas posibles de memoria.

ROUTER

Router.events proporciona eventos como observables. Puede usar el operador filter() de RxJS para buscar eventos de interés y suscribirse a ellos para tomar decisiones basadas en la secuencia de eventos en el proceso de navegación.

ActivedRoute es un servicio de enrutador inyectado que utiliza observables para obtener información sobre una ruta y parámetros.

REACTIVE FORMS

Tienen propiedades que usan observables para monitorear los valores de control de formas. Las propiedades FormControl, valuesChanges y statusChanges contienen observables que generan eventos de cambio. La suscripción a una propiedad de control de forma observable es una forma de activar la lógica de la aplicación dentro de la clase de componente.

ANGULAR ROUTER

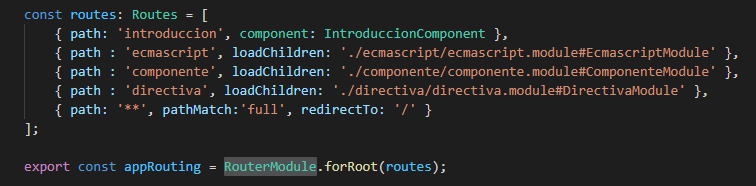
Con Angular construimos aplicaciones de una sóla página (SPA, Single Page Application). Esto consiste en mostrar todas las vistas o pantallas en una única pagina, sin realizar sucesivas cargas de otras direcciones provenientes del servidor.

Es decir, el usuario puede cambiar lo que ve, mostrando u ocultando partes de la pantalla que corresponden a componentes particulares, en lugar de solicitarlos al servidor.

Para poder manejar la navegación de una vista a otra se utiliza a Angular Router, un enrutador del lado cliente. El cual permite la navegación interpretando la URL del navegador como una instrucción para cambiar la vista.

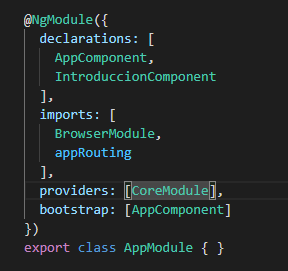
CONFIGURAR ANGULAR ROUTER

Para crear el enrutamiento en Angular hace falta crear un archivo convencionalmente llamada app-routing.module.ts, el cual contendrá las rutas deseadas en forma de array.



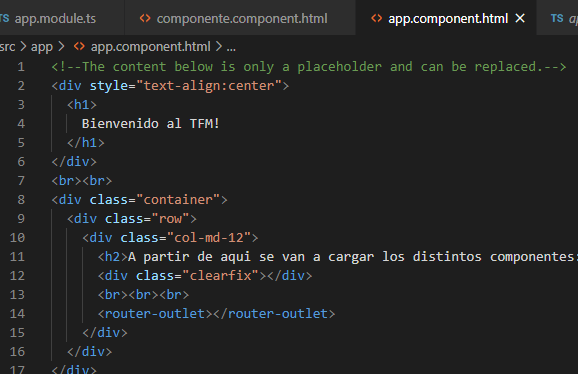
En el ejemplo anterior se observan dos formas de definir las rutas. Definiendo directamente una ruta a un componente concreto y definiendo una ruta a un módulo en concreto que tendrá su propia configuración de enrutamiento.

El siguiente paso es importar la constante creada en nuestro módulo raíz.



Además debemos añadir el elemento <base href=”/> a la pagina principal de la aplicación, nuestro fichero index.html, para indicar al enrutador cómo componer las URL de navegación.

Por último, para tener un enrutamiento básico debemos agregar la directiva routerOutlet a nuestro componente principal, pues así le decimos a Angular que todos los componentes se localizarán donde colocamos la directiva mencionada.



ANGULAR Y PRINCIPIOS DE DISEÑO

ANGULAR Y EL PRINCIPIO DE RESPONSABILIDAD ÚNICA

ANGULAR Y EL PRINCIPIO ABIERTO-CERRADO

ANGULAR Y EL PRINCIPIO DE SUSTITUCIÓN DE LISKOV

Resumen y Conclusión

**Resumen del trabajo en un máximo de cien (100) palabras.**

En este último capítulo se realizará un balance de los resultados obtenidos después de la realización del Proyecto Fin de Carrera…

## Conclusiones y Futuras Líneas de Trabajo

**Conclusiones y futuro trabajo.**

Después de dar por finalizado este Proyecto Fin de Carrera “Accesibilidad en la Web: Normas y Aplicación”, hacemos balance sobre los conocimientos adquiridos y las impresiones obtenidas a lo largo del mismo…

Bibliografía

**Bibliografía, donde se indicará el conjunto de las referencias utilizadas como citas y otros materiales de consulta, siempre que, a lo largo del trabajo, se afirma algo que no se demuestra, conteniendo cada una los siguientes datos:**

**- Autor/es.**

**- Título del artículo, libro, monografía,...**

**- Editorial o nombre de la revista y editorial.**

**- Número de la revista, volumen y páginas consultadas.**

**- Año de publicación.**

**- Asimismo, en este apartado se reseñarán los distintos catálogos utilizados.**

1. Abascal, J., Valero, P. (2001), Curso Introducción a la Interacción Persona-Ordenador: El libro electrónico, AIPO, <http://griho.udl.es/ipo/libroe.html>
2. Asociación Española de Normalización y Certificación (1986), Normas y Publicaciones <http://www.aenor.es/>