

# Curso Programación Front

Septiembre 2022

José María González  
[consultas@avante.es](mailto:consultas@avante.es)

[www.avante.es](http://www.avante.es)   [avante@avante.es](mailto:avante@avante.es)   902 117 902

**avante**  
¿hasta dónde quieras llegar?



## PRESENTACIÓN

- Sobre el curso
- Presentación
- Alcance
- Metodología
- Descansos

## ÍNDICE PARTE 1

- Tema 1: Introducción a la programación Frontend
- Tema 2. jQuery
- Tema 3. Frameworks de desarrollo HTML - Bootstrap
- Tema 4. SASS



## TEMA 1: INTRODUCCIÓN A LA PROGRAMACIÓN FRONTEND

Algunos factores han hecho que cambie el paradigma de programación frontend

- Cambio de paradigma UI/UX
- Herramientas de protipado rápido
- Metodologías ágiles
- Frameworks de desarrollo frontend

## TEMA 1: INTRODUCCIÓN A LA PROGRAMACIÓN FRONTEND

También afectan a la programación Backend

- Irrupción de la demanda orientadas a multidispositivo
- Interacción entre sistemas gracias a sistemas SOA y generalización de servicios web
- Arquitectura microservicios

## TEMA 1: INTRODUCCIÓN A LA PROGRAMACIÓN FRONTEND

Estrategias de renovación de legacy software o sistemas heredados

- Mantenimiento
- Reescritura del código
- SOA y Servicios web
- Modernización de frontends y mejora de la separación de capas

## TEMA 1: INTRODUCCIÓN A LA PROGRAMACIÓN FRONTEND

La irrupción de HTML 5 ha supuesto un esfuerzo de estandarización y mejora de la web

- Uniformidad en el estándar para mejorar la compatibilidad (aunque no del todo)
- Nuevas estructuras semánticas para mejorar la estructura de la web
  - <article>, <footer>, <main>, ...
- Nuevos elementos de formularios
  - <output> y <datalist>
- Nuevos input types
  - <time>, <email>, ...
- Aparición del canvas
- Aparición de nuevos elementos para embeber medios

## TEMA 1: INTRODUCCIÓN A LA PROGRAMACIÓN FRONTEND

### Herramientas para el curso

- Por un lado veremos herramientas y frameworks orientados a mejorar la experiencia de nuestro sitio y que son fáciles de integrar:
  - jQuery
  - Bootstrap
  - SASS
  - Gulp
- En la segunda parte veremos un framework orientado a Frontend:
  - Typescript
  - Angular



## TEMA 2: JQUERY

jQuery supuso una pequeña revolución en el mundo HTML

- Herramienta sencilla de uso orientada al manejo del DOM y de CSS
- Compatible para múltiple navegadores
- Con bajo nivel de incompatibilidad
- Ideal para expandir la funcionalidad de nuestro frontend
- Para usarlo tan solo tenemos que importarlo en nuestra web o usar un CDN
  - <http://jquery.com/download/>
  - <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>



## TEMA 2: JQUERY

### Elementos principales

- Selector del DOM
  - \$("button")
  - \$("#id")
  - \$(".class")
- Gestión de eventos
  - \$(".click")
  - \$(".on("click", function(){});
- Llamadas AJAX
  - ```
$ajax({
    url: "/api/getWeather",
    data: {
        zipcode: 97201
    },
    success: function( result ) {
        $("#weather-temp").html( "<strong>" + result + "</strong> degrees" );
    }
});
```

## TEMA 2: JQUERY

### Selectores

- Los selectores nos permiten acceder a elementos del DOM
- Para asegurarnos que el DOM se ha cargado usamos la función `$(document).ready(function(){...});`
- Normalmente usamos un callback para ejecutar lo que queramos.
- Selectores
  - `$("button")`
  - `$("#id")`
  - `$(".class")`
  - `$("[name='miNombre'])"`
  - `$(“ul > li”)`
  - `http://api.jquery.com/category/selectors/`

## TEMA 2: JQUERY

Una vez hemos seleccionado nuestro elemento podemos ejecutar acciones

- Hay acciones ya creadas
  - Change, click, dblclick, hover...
- Pero también podemos tomar el control sobre el evento, por ejemplo deteniendo la acción por defecto de un elemento
  - Event.eventPreventDefault()
- También podemos asociar eventos con on para un evento o bind para varios eventos personalizados y también los normales.
  - `$( "#miid" ).bind{click: function() {...},mouseenter: function() {...}};`
  - `$( "#miid" ).on( "click", function() {...});`
- <http://api.jquery.com/category/events/>

## TEMA 2: JQUERY

Podemos modificar nuestros objetos del DOM

- A través de los atributos
  - addClass(), hasClass(), attr(), html(), val(),...
- También modificar el DOM directamente
  - append(), prepend(), text(), before(), after(), remove(),
- Al seleccionar un elemento podemos iterar en sus elementos
  - each(), first(), last(), parent(), children()
  - Podemos usar \$(this) para referirnos al elemento actual.

## TEMA 2: JQUERY

También podemos hacer algunos efectos

- Mostrar u ocultar elementos
  - Hide(), show(), toggle()
- Animarlos
  - animate(), fadeIn, fadeOut(), slideIn()...
- <http://api.jquery.com/category/effects/>

## TEMA 2: JQUERY

Podemos hacer llamadas a servicios web de manera sincrona y asíncrona con Ajax

- Podemos usar funciones para hacer peticiones
  - get(), post()
  - Estas funciones devuelven un objeto (data por convención) al que podemos acceder.
  - También podemos generar una función callback para tratar el error.
- Para peticiones asíncronas podemos usar \$.ajax

```
// Assign handlers immediately after making the request,
// and remember the jqXHR object for this request
var jqxhr = $.ajax( "example.php" )
  .done(function() {
    alert( "success" );
  })
  .fail(function() {
    alert( "error" );
  })
  .always(function() {
    alert( "complete" );
  });

// Perform other work here ...

// Set another completion function for the request above
jqxhr.always(function() {
  alert( "second complete" );
});
```

## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

¿Para que sirven los frameworks de desarrollo web?

- Nos permiten generar nuestro frontend resolviendo problemas
  - Adaptación a diversas pantallas
  - Componentes reusables que nos permiten generar código rápidamente y con un aspecto uniforme.
- Bootstrap nos ofrece además
  - Adaptabilidad a través de SASS para generar hojas de estilos rápidamente.
  - Librerías para interactuar a través de jQuery
  - Themes ya generados que nos permiten empezar directamente a trabajar en nuestro frontend.
- ¿Cómo se utiliza?
  - Instalando la librería CSS y las dependencias de JavaScript
    - <https://getbootstrap.com/>
- ¡¡¡BAJO NINGÚN CONCEPTO DEIS SOPORTE A IE9!!!

## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

Para trabajar un documento necesitamos algunos elementos

- Tenemos que señalar nuestro documento como HTML5
  - <!doctype html> <html lang="en">
- Tenemos que poner una etiqueta para indicar que es responsive
  - <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
- <https://getbootstrap.com/docs/4.1/getting-started/introduction/#starter-template>

## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

### Gestión del layout

- Podemos generar nuestro contenido de manera muy sencilla utilizando flexbox.

```
<div class="container">
  <div class="row">
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
  </div>
</div>
```

## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

### Gestión del layout

- La división se produce a doce columnas y estas son las opciones disponibles.

	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	Extra large ≥1200px
<b>Max container width</b>	None (auto)	540px	720px	960px	1140px
<b>Class prefix</b>	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-
<b># of columns</b>	12				
<b>Gutter width</b>	30px (15px on each side of a column)				
<b>Nestable</b>	Yes				
<b>Column ordering</b>	Yes				



## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

### Gestión del layout

- La división se produce a doce columnas y estas son las opciones disponibles.

	<b>Extra small</b> $<576\text{px}$	<b>Small</b> $\geq 576\text{px}$	<b>Medium</b> $\geq 768\text{px}$	<b>Large</b> $\geq 992\text{px}$	<b>Extra large</b> $\geq 1200\text{px}$
<b>Max container width</b>	None (auto)	540px	720px	960px	1140px
<b>Class prefix</b>	<code>.col-</code>	<code>.col-sm-</code>	<code>.col-md-</code>	<code>.col-lg-</code>	<code>.col-xl-</code>
<b># of columns</b>	12				
<b>Gutter width</b>	30px (15px on each side of a column)				
<b>Nestable</b>	Yes				
<b>Column ordering</b>	Yes				

- En la documentación tenemos más formas de organizar nuestro layout. Normalmente se suele coordinar con un diseñador gráfico

## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

### Creación de contenidos a través de Media Objects

- Permite generar contenido con tan solo dos etiquetas que podemos reutilizar para generar nuestro contenido
- Podemos ver todas las opciones
  - <https://getbootstrap.com/docs/4.1/layout/media-object/>

#### Media heading

64x64

Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin. Cras purus odio, vestibulum in vulputate at, tempus viverra turpis. Fusce condimentum nunc ac nisi vulputate fringilla. Donec lacinia congue felis in faucibus.

```
<div class="media">
  
  <div class="media-body">
    <h5 class="mt-0">Media heading</h5>
    Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin. Cras
  </div>
</div>
```

Copy

## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

### Elementos de HTML5

- Bootstrap da forma a muchas de las etiquetas de HTML5 para darle un estilo más limpio, así como atributos para gestionar eventos normales o para definir la tipografía
  - <https://getbootstrap.com/docs/4.1/content/reboot/>
- También nos ofrece estilos para tratar nuestras imágenes.
  - <https://getbootstrap.com/docs/4.1/content/images/>
- Otro elemento interesante son las tablas
  - <https://getbootstrap.com/docs/4.1/content/tables/>



## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

### Componentes

- Bootstrap tiene un amplio catalogo de componentes que podemos usar
  - Alertas
  - Badges
  - Miga de pan
  - Botones
  - Modales
  - ...
- <https://getbootstrap.com/docs/4.1/components/alerts/>

## TEMA 3: FRAMEWORKS DE DESARROLLO HTML - BOOTSTRAP

### Javascript y Bootstrap

- Bootstrap tiene funciones específicas para controlar componentes o reaccionar ante eventos.
- A través de jQuery, como hemos visto en los ejemplos, podemos añadir o modificar clases para modificar nuestro layout.

## TEMA 4: SASS

¿Cómo desarrollar con SASS?

- SASS es una utilidad que permite compilar hojas de estilo asignando variables.
- Esto permite adaptar nuestras hojas de estilo según nuestras necesidades, por ejemplo para personalizar una aplicación para un cliente.
- También nos puede servir para reutilizar nuestra aplicación.
- SASS nos ayuda también a la hora de poder diferenciar nuestra hoja de estilos y agruparla después para tener una única hoja de estilos.
- Para instalarla usamos NPM
  - `npm install -g sass`
- Para ejecutarlo usamos la linea de comando
  - `sass fichero.scss fichero.css`

## TEMA 4: SASS

¿Cómo desarrollar con SASS?

- Para generar nuestra CSS usamos variables en SASS.
  - `$primary-color: #333;`
  - `color: $primary-color;`
- También podemos anidar nuestros estilos
  - `nav { ul { margin: 0; padding: 0; list-style: none; }}`
- Podemos crear ficheros parciales llamandolos con `_nombre.scss`
- Luego los podemos importar en otros scss
  - `@import 'nombre';`
- También contemplan herencia
  - `%mi-estilo{ display: none; }`
  - `.message { @extend % mi-estilo; }`

## ÍNDICE PARTE 2

- Tema 1: Introducción a Angular 13
- Tema 2. JavaScript
- Tema 3. TypeScript
- Tema 4. Componentes
- Tema 5. Directivas
- Tema 6. Routing
- Tema 7. Servicios
- Tema 8. Formularios
- Tema 9. Pipes
- Tema 10. HTTP
- Tema 11. Testing

## TEMA 1: INTRODUCCIÓN A ANGULAR 13

Veremos las siguientes cuestiones

- Que es Angular y que cambios se han producido con respecto a la versión 1.0
- Cuales son las diferentes partes de Angular y sus componentes
- Como se estructura un proyecto de Angular
- Instalación con npm
- Introducción a Angular CLI

## TEMA 1: INTRODUCCIÓN A ANGULAR 13

- Angular es un framework orientado a componentes
  - Los componentes pueden a su vez tener subcomponentes y así sucesivamente.
- Angular incorpora plantillas de declaraciones donde podemos declarar elementos como etiquetas de HTML que incorporan componentes de Angular en nuestro HTML.
- Angular también tiene inyección de dependencias
- Incluye el Router para ayudar al cliente a navegar en la aplicación
- Integra RxJS para resolver llamadas asincronas entre otras funcionalidades
- Para escribir en Angular podemos usar Typescript o Javascript entre otros

## TEMA 1: INTRODUCCIÓN A ANGULAR 13

### Mejoras con respecto a Angular JS

- Se mejora el rendimiento para páginas más complejas. Ahora no se guardan todas las variables en el \$scope y el \$watch es sustituido por observables que son mucho más eficientes
  - \$watch daba bastantes problemas a la hora de evaluar páginas muy complejas, causando incluso problemas de rendimientos.
- Se cambia de un paradigma MVC a una orientación por componentes, mucho más natural a la hora de desarrollar para frontend.
- AngularJS no estaba pensado para su uso en móvil, en cambio Angular.io si lo está. Los componentes mejoran su uso con respecto a AngularJS
- Ya no se da soporte a AngularJS, si tenemos una aplicación viva conviene ir preparando la migración a Angular 13



## TEMA 1: INTRODUCCIÓN A ANGULAR 13

### Sobre Angular 2.0 y posteriores

- En angular.io podremos encontrar toda la documentación sobre Angular a partir de su versión 2.0
- Para la versión anterior se sigue usando angularjs.org
- Angular 2 es una reescritura completa de angular. Se cambia el foco de una arquitectura MVC a una basada en servicios y controladores.
- Cada nueva versión implica una reducción del bundle del angular para reducir la descarga de la aplicación
- Un cambio sustancial que ha mejorado la robustez es el uso de Typescript



## TEMA 1: INTRODUCCIÓN A ANGULAR 13

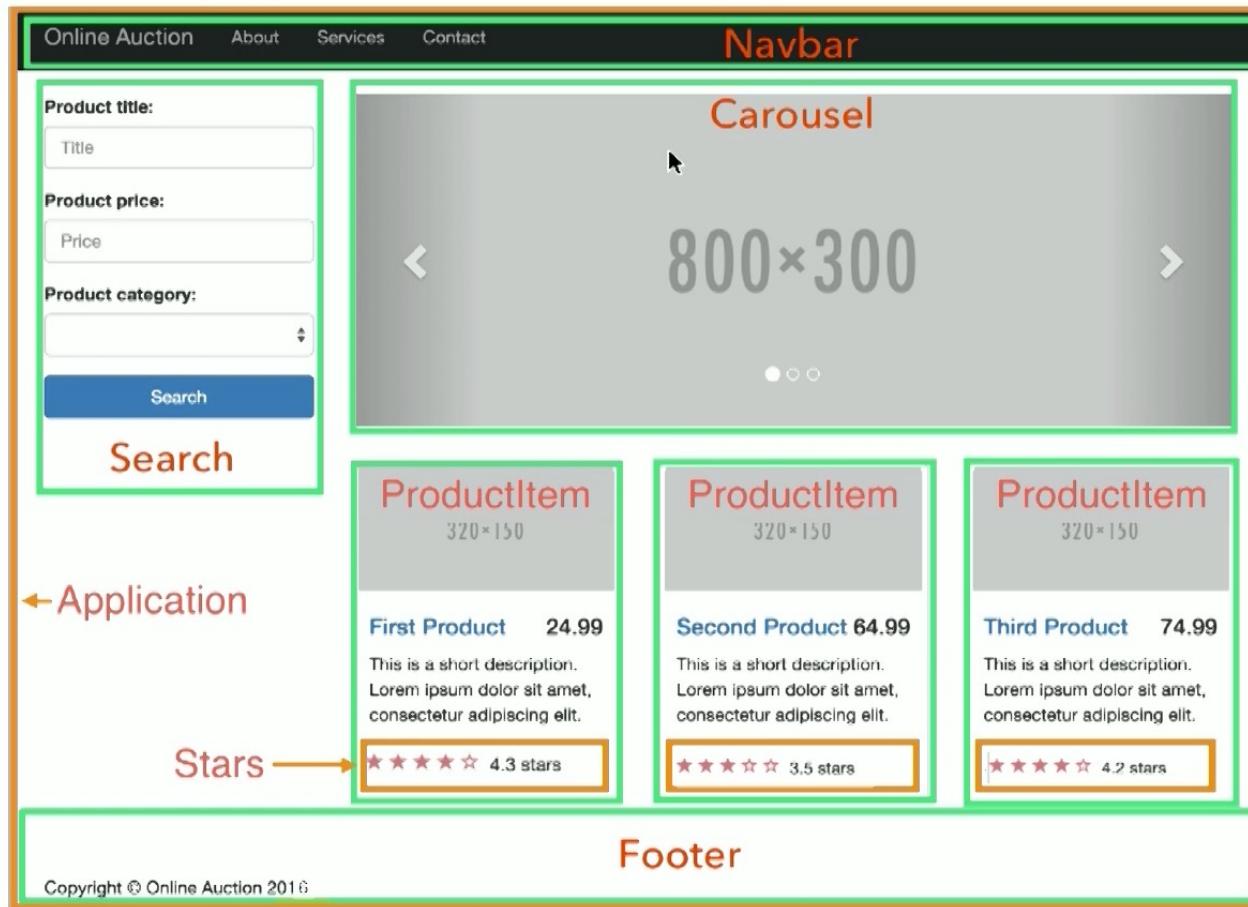
- Estos son los elementos principales de Angular
  - Componente: es una clase que incluye un UI (una plantilla)
  - Directiva: Una clase con código para realizar acciones en componentes pero que no tienen UI de por si.
  - Servicio: Una clase que contiene la lógica de negocio de la aplicación que se inyectan en los componentes.
  - Tubería o Pipe: una función transformadora que se usa en las plantillas. Un buen ejemplo sería una tubería para formatear fechas o monedas.
  - Módulo: una clase que contiene listas de componentes, directivas, servicios, otros módulos o pipes. Una app mínima es un módulo de Angular. También es la unidad mínima para el despliegue.

## TEMA 1: INTRODUCCIÓN A ANGULAR 13

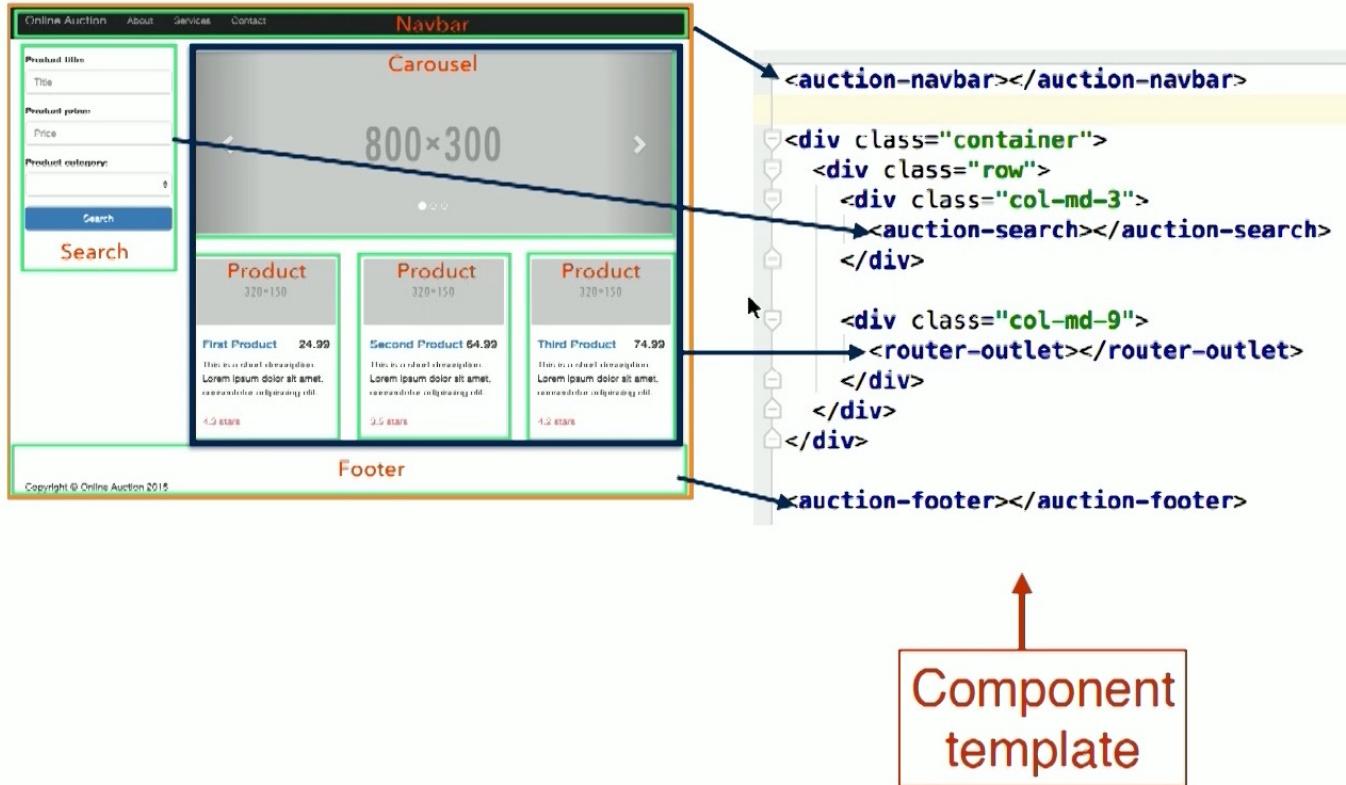
### Como migrar a Angular 13

- En general no hay una manera directa de migrar una aplicación de angularJS a angular.io
- Una estrategia de migración suele ser intentar modularizar al máximo la aplicación:
  - Crear un modelo de datos coherente con la nueva aplicación
  - Separar los servicios en clases propias
  - Separar y modularizar al máximo los componentes estructurales
  - Es posible reutilizar el código HTML, aunque tendremos que cambiar bastantes etiquetas de angular ya que la notación ha cambiado

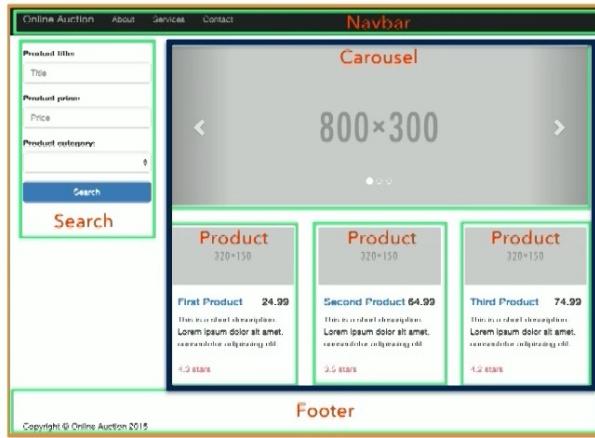
# TEMA 1: INTRODUCCIÓN A ANGULAR 13



# TEMA 1: INTRODUCCIÓN A ANGULAR 13



# TEMA 1: INTRODUCCIÓN A ANGULAR 13



```

<auction-navbar></auction-navbar>

<div class="container">
  <div class="row">
    <div class="col-md-3">
      <auction-search></auction-search>
    </div>

    <div class="col-md-9">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>

<auction-footer></auction-footer>

```

```

import {Component} from '@angular/core';
import {Product, ProductService} from './product.service';

@Component({
  selector: 'app-root',
  templateUrl: 'application.html',
  styleUrls: ['application.css']
})
export class AppComponent {
  products: Array<Product> = [];

  constructor(private productService: ProductService) {
    this.products = this.productService.getProducts();
  }
}

```

HTML and CSS

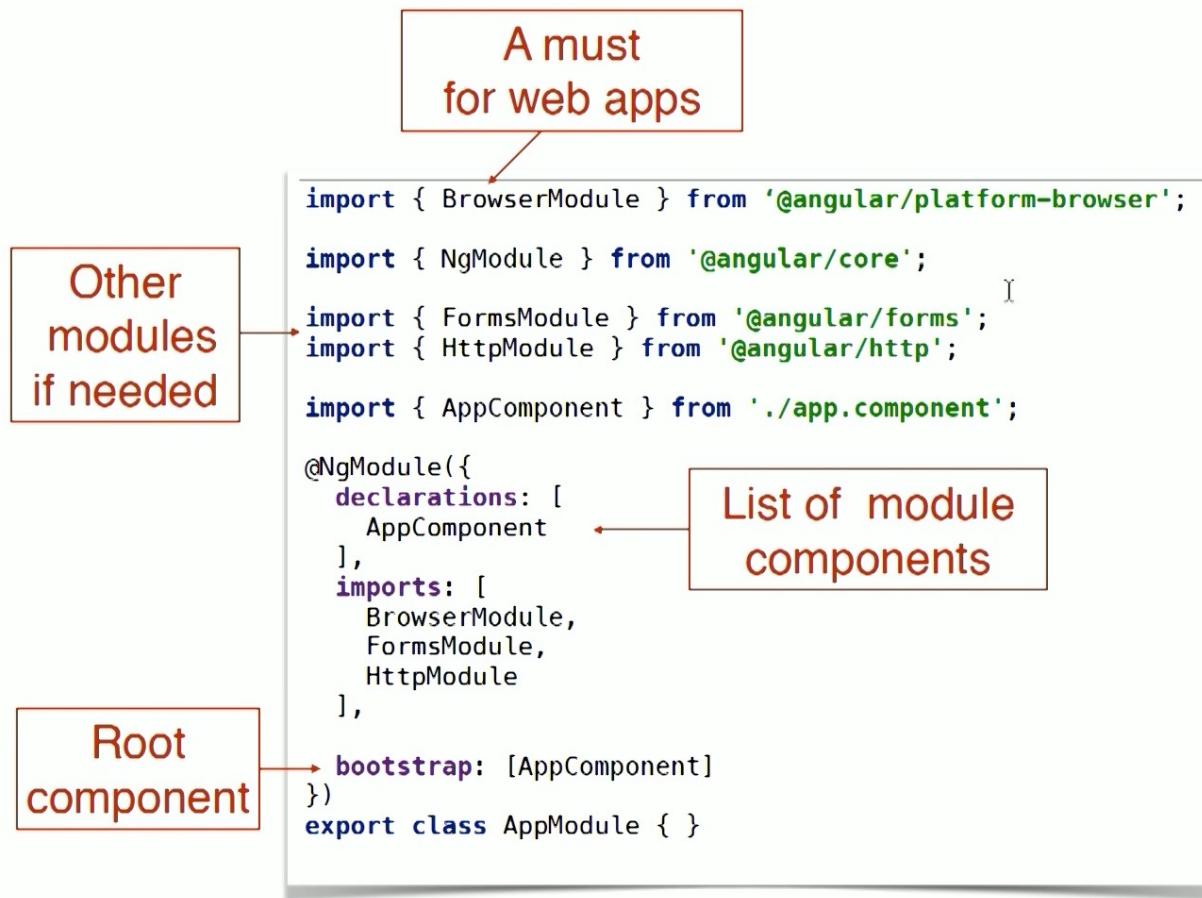
TypeScript

## TEMA 1: INTRODUCCIÓN A ANGULAR 13

Un módulo de angular

- Una clase con el decorador `@NgModule`
- Es una unidad de compilación y distribución
- Lista todos los componentes, servicios, routes y otros
- Una aplicación debe contener al menos un módulo (una clase con `@NgModule`)

## TEMA 1: INTRODUCCIÓN A ANGULAR 13



## TEMA 1: INTRODUCCIÓN A ANGULAR 13

- NPM es una herramienta de instalación de paquetes con dependencias tipo maven.
- Vamos a crear nuestro primer proyecto para ello instalaremos npm
  - <https://www.npmjs.com/package/npm>
- Una vez instalado npm podemos usar Angular CLI. Angular CLI no es más que un interfaz de línea de comando específico para proyectos con Angular.
- Podemos instalarlo con esta línea
  - `npm install -g @angular/cli`
- Una vez instalado podemos crear proyectos, generar componentes, servir la aplicación y más funcionalidades que veremos a lo largo del curso.



## TEMA 2: JAVASCRIPT

- Veremos las siguientes cuestiones
  - Javascript y Typescript
  - Diferencias entre versiones de Javascript ECMA 5 – ECMA 6

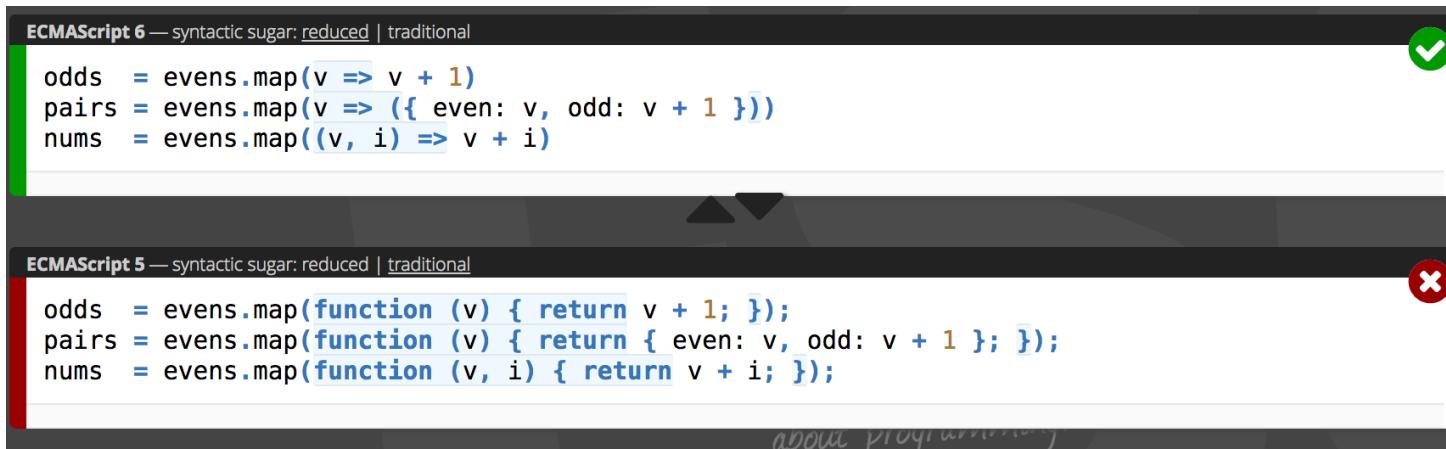
## TEMA 2: JAVASCRIPT

- En Angular tenemos la posibilidad de seleccionar para que tipo de Javascript queremos que funcione nuestra aplicación. Normalmente trabajamos con ECMA 6, pero es importante saber que podemos adaptarnos ya que según que proyectos es posible que tengamos que bajar de versión.
- Recordemos que Typescript no es más que una capa superior de Javascript, por lo que realmente todo el código que generamos al final el servidor lo traduce a Javascript.
- Angular hace una traducción del código a la versión de Javascript y lo que ejecuta el navegador es Javascript puro.

## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Constantes
  - ECMA 6 permite definir constantes `const PI = 3.1415`
- Scope
  - Mejora en el scope de variables, permite redefinir variables para un scope
  - También mejora en el scope de funciones.
- Arrow functions
  - Mejoras en las expresiones arrow (parecidas a las lambdas de Java)



ECMAScript 6 — syntactic sugar: reduced | traditional ✓

```
odds  = evens.map(v => v + 1)
pairs = evens.map(v => ({ even: v, odd: v + 1 }));
nums  = evens.map((v, i) => v + i)
```

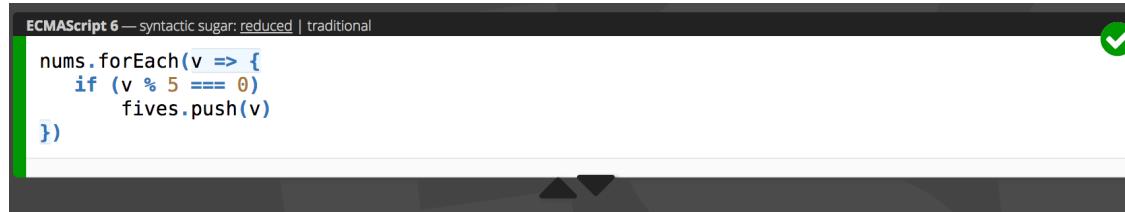
ECMAScript 5 — syntactic sugar: reduced | traditional ✗

```
odds  = evens.map(function (v) { return v + 1; });
pairs = evens.map(function (v) { return { even: v, odd: v + 1 }; });
nums  = evens.map(function (v, i) { return v + i; });
```

## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Arrow functions
  - Mejoras en las expresiones arrow (parecidas a las lambdas de Java)

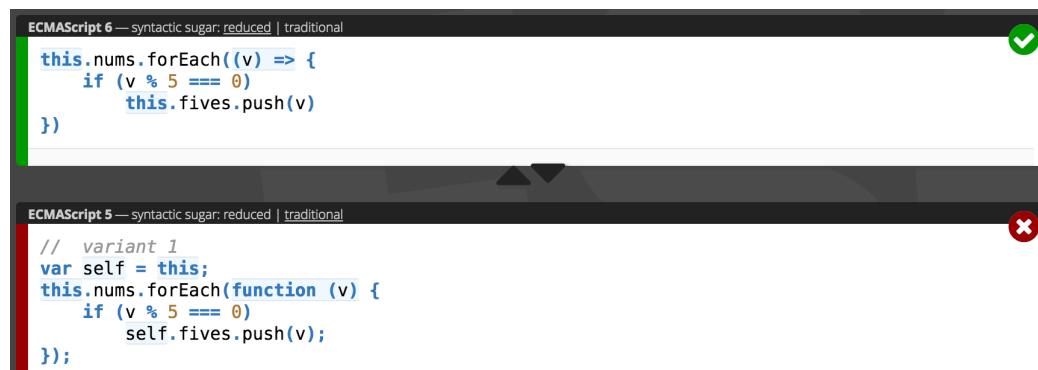


ECMAScript 6 — syntactic sugar: reduced | traditional

```
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v)
})
```

ECMAScript 5 — syntactic sugar: reduced | traditional

```
nums.forEach(function (v) {
  if (v % 5 === 0)
    fives.push(v);
});
```



ECMAScript 6 — syntactic sugar: reduced | traditional

```
this.nums.forEach((v) => {
  if (v % 5 === 0)
    this.fives.push(v)
})
```

ECMAScript 5 — syntactic sugar: reduced | traditional

```
// variant 1
var self = this;
this.nums.forEach(function (v) {
  if (v % 5 === 0)
    self.fives.push(v);
});
```

## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Mejoras en la gestión de parámetros
  - Mejora en parámetros undefined



The screenshot shows two side-by-side browser developer tool panes comparing ECMAScript 6 and ECMAScript 5 code execution results.

**ECMAScript 6 — syntactic sugar: reduced | traditional**

```
function f (x, y = 7, z = 42) {
  return x + y + z
}
f(1) === 50
```

**ECMAScript 5 — syntactic sugar: reduced | traditional**

```
function f (x, y, z) {
  if (y === undefined)
    y = 7;
  if (z === undefined)
    z = 42;
  return x + y + z;
}
f(1) === 50;
```

A green checkmark icon is present in the top right corner of the ECMAScript 6 pane, while a red X icon is present in the top right corner of the ECMAScript 5 pane.

## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Mejoras en la gestión de parámetros
  - También de manera similar en llamadas rest y en el operador spread
    - `var params = [ "hello", true, 7 ]`
    - `var other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]`
- Mejoras en plantillas de literales
  - Permite de una manera más sencilla extrapolar cadenas, llamadas a métodos y acceso a literales.
- Extensión de literales
  - Para binarios y octales
    - `0b111110111 === 503`
    - `0o767 === 503`
  - Para Unicode



## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Mejoras en las expresiones regulares
  - Recuerda la última posición del match para no tener que reevaluar la misma expresión varias veces.
- Mejoras en las propiedades de los objetos
  - ECMA 6: obj = { x, y }
  - ECMA 5: obj = { x: x, y: y };
  - También permite atributos con nombres calculados
    - `let obj = { foo: "bar", [ "baz" + quux() ]: 42 }`
  - Permite también definir funciones de una manera más sencilla en objetos.
    - `obj = { foo (a, b) { ... }`

## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Mejoras en la desestructuración de asignaciones
  - En arrays permite desctructurarlos más facilmente
    - `var list = [ 1, 2, 3 ]; var [ a, , b ] = list; [ b, a ] = [ a, b ]`
  - En objetos
    - `var { op, lhs, rhs } = getASTNode()`
    - `var { op: a, lhs: { op: b }, rhs: c } = getASTNode()`
  - También mejora la asignación por defecto y con undefined al desestructurar objetos y arrays
- Mejoras en módulos
  - Soporta exportar funcionalidad en módulos e importarlos de una manera más sencilla a través de `export` e `import`
    - `export function sum (x, y) { return x + y } // lib/math.js`
    - `import * as math from "lib/math" // someApp.js`

## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Permite el uso de clases de una manera más natural
  - `class Shape { constructor (id, x, y) { this.id = id this.move(x, y) } move (x, y) { this.x = x this.y = y } }`
  - Usa extends para indicar herencia con super para acceder al parente
  - Uso de estáticos como en Java a través de static
- Introducción de Symbol
  - Para crear objetos inmutables para ser usados como identificadores de propiedades de objetos
  - También a nivel de aplicación
    - `Symbol.for("app.foo")`
- Mejoras en iteradores para hacerlos más similares a los de Java
  - `for (let n of fibonacci) { ... }`



## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

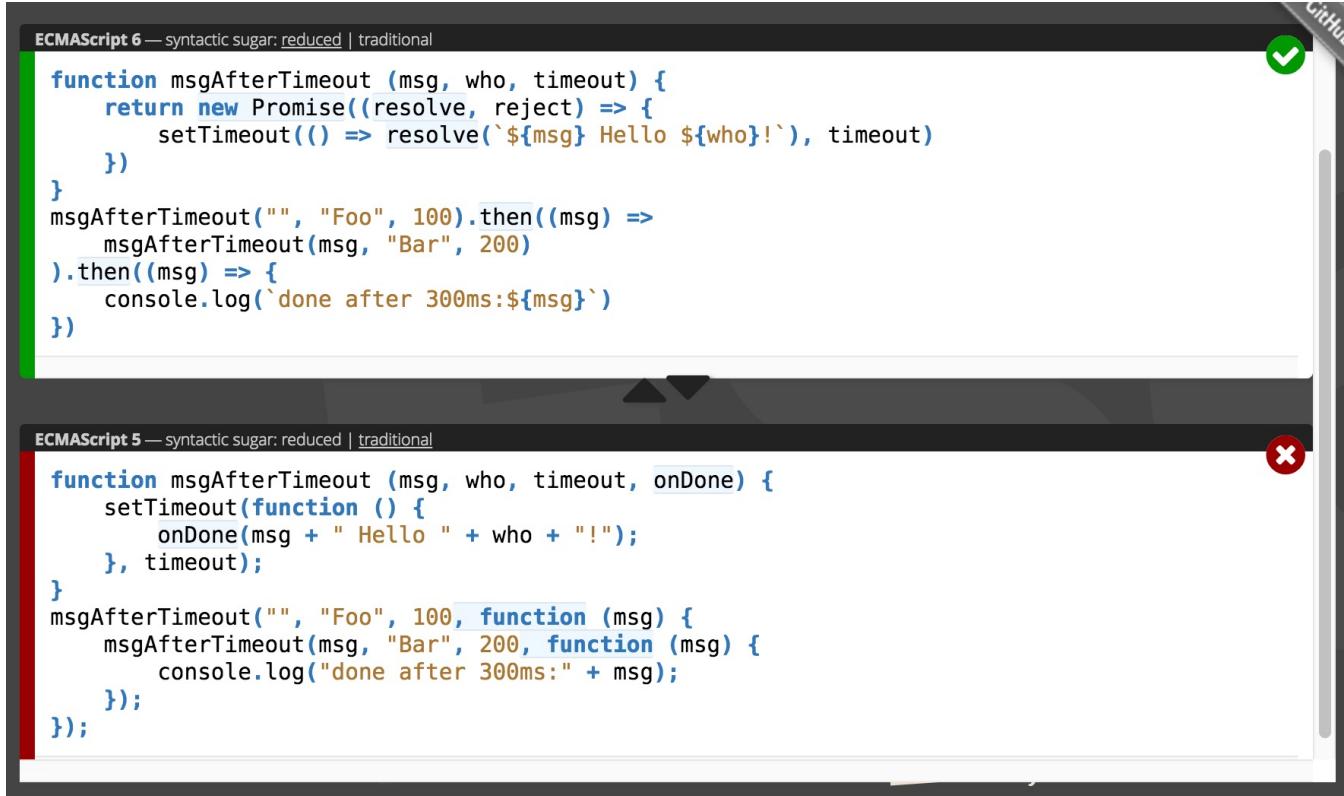
- Mejoras en mapas y conjuntos
  - Introduce Set() con add() y has(), entre otros, en lugar de arrays
  - Introduce Map() con set() y get(), entre otros, para mapas con clave/valor
- Permite crear arrays tipados
  - **new Uint8Array (...)**
- Nuevas funcionalidades en objetos
  - Assign para asignación a objetos
    - Object.assign(dest, param1, param2, ...)
  - find y findIndex para buscar en arrays
  - repeat para repetir cadenas
  - Más funcionalidad sobre cadenas como startsWith, endsWith, includes
  - isNaN, isFinite para números
  - isSafeInteger para comprobación de tipos
  - Math.trunc para truncar números
  - Math.sign para determinar el signo de números



## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Mejora en la promesas que además permite combinar varias



The screenshot shows two code snippets side-by-side in a code editor.

**ECMAScript 6 — syntactic sugar: reduced | traditional**

```
function msgAfterTimeout (msg, who, timeout) {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(`[${msg}] Hello ${who}!`), timeout)
    })
}
msgAfterTimeout("", "Foo", 100).then((msg) =>
    msgAfterTimeout(msg, "Bar", 200)
).then((msg) => {
    console.log(`done after 300ms:${msg}`)
})
```

**ECMAScript 5 — syntactic sugar: reduced | traditional**

```
function msgAfterTimeout (msg, who, timeout, onDone) {
    setTimeout(function () {
        onDone(msg + " Hello " + who + "!");
    }, timeout);
}
msgAfterTimeout("", "Foo", 100, function (msg) {
    msgAfterTimeout(msg, "Bar", 200, function (msg) {
        console.log("done after 300ms:" + msg);
    });
});
```

The ECMAScript 6 code is highlighted with a green background and has a green checkmark icon in the top right corner. The ECMAScript 5 code is highlighted with a red background and has a red X icon in the top right corner.

## TEMA 2: JAVASCRIPT

### Diferencias entre ECMA 5 y ECMA 6

- Introducción de la internacionalización
  - Para la declaración y comparación
    - **new Intl.Collator("de")**
  - Para el formateo de números
    - **Intl.NumberFormat("en-US")**
  - Para el formateo de moneda
    - **new Intl.NumberFormat("en-US", { style: "currency", currency: "USD" })**
  - Y para el formateo de fecha y hora
    - **new Intl.DateTimeFormat("en-US")**

## TEMA 3: TYPESCRIPT

- Veremos las siguientes cuestiones
  - Introducción a TS
  - Programación orientada a componentes
  - Angular 6 en TS
- También haremos nuestros primeros ejercicios en Typescript con Angular.

## TEMA 3: TYPESCRIPT

- **¿Que es TypeScript? :**
  - Código abierto superconjunto de JavaScript desarrollado por Microsoft
  - Compila código en JavaScript de varios ECMAScript
  - Bien soportado por IDEs
  - Página oficial: <http://www.typescriptlang.org>
- **¿Por que TypeScript? :**
  - Opcional el especificar un tipo de variable
  - Soporta las últimas características de JavaScript
  - Más productivo que JavaScript
  - Soporta clases,interfaces,anotaciones genéricas, modificadores de acceso (public/private/protected) y más.

## TEMA 3: TYPESCRIPT

- **Instalar TypeScript :**
  - npm install typescript -g <- instalar compilador de TypeScript global
  - ng new (app name) <- Crear nueva aplicación
  - ng serve --app (app name) <- iniciar aplicación específica
- Vamos a crear un proyecto nuevo y analizaremos el mismo.

## TEMA 3: TYPESCRIPT

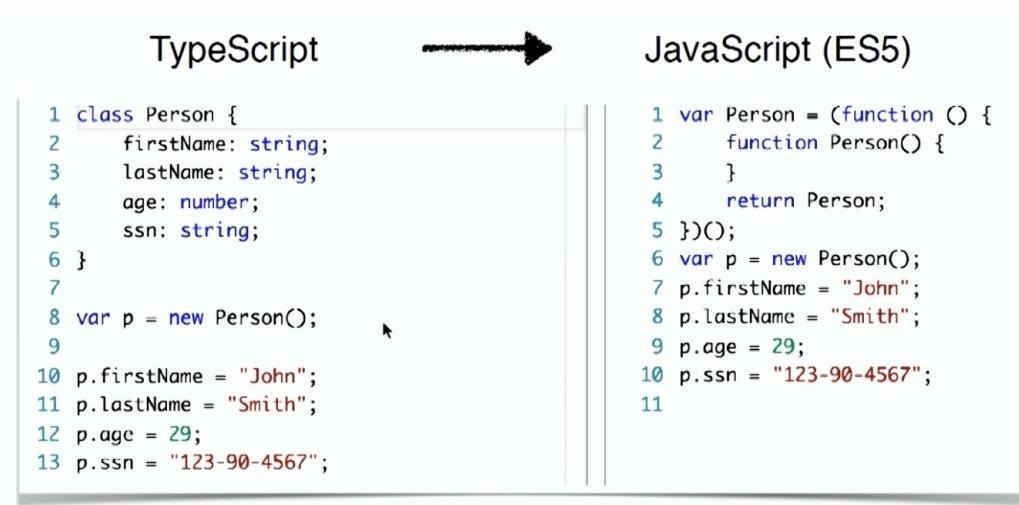
### Variables en Typescript

- Con TypeScript podemos usar las novedades para JavaScript que nos trae el ES6, podemos definir variables de dos formas, utilizando **var** como ya veníamos utilizando en JavaScript clásico y también podemos usar una nueva palabra reservada que es **let**.
- La diferencia es el alcance de las variables.
  - **let** : permite declarar variables limitando su alcance al bloque, declaración, o expresión donde se está usando.
  - **var** : define una variable global o local en una función sin importar el ámbito del bloque.

## TEMA 3: TYPESCRIPT

### Clases en Typescript

- En TypeScript podemos hacer uso de clases, como en el siguiente ejemplo:



The diagram illustrates the conversion of a TypeScript class declaration into its equivalent in JavaScript (ES5). On the left, under 'TypeScript', is the following code:

```
1 class Person {  
2     firstName: string;  
3     lastName: string;  
4     age: number;  
5     ssn: string;  
6 }  
7  
8 var p = new Person();  
9  
10 p.firstName = "John";  
11 p.lastName = "Smith";  
12 p.age = 29;  
13 p.ssn = "123-90-4567";
```

An arrow points from this code to the right side, labeled 'JavaScript (ES5)', which contains:

```
1 var Person = (function () {  
2     function Person() {  
3     }  
4     return Person;  
5 })();  
6 var p = new Person();  
7 p.firstName = "John";  
8 p.lastName = "Smith";  
9 p.age = 29;  
10 p.ssn = "123-90-4567";  
11
```

- La parte izquierda de la imagen sería un ejemplo de declaración de una clase “Person” con TypeScript, y la parte de la derecha sería su equivalente en JavaScript (ES5).
- En JavaScript(ES5) no usamos la palabra “class” , puesto que (ES5) no soporta clases. Sin embargo, en JavaScript(ES6) si las soporta.

## TEMA 3: TYPESCRIPT

# Arrow Functions

- En TypeScript podemos hacer uso de arrow functions (parecidas a lambdas de Java):

```
let getName = () => 'John Smith';  
console.log(`The name is ` + getName());
```

- En color rojo se puede apreciar una expresión “Arrow function”, es una alternativa para escribir funciones anónimas en JavaScript.
  - En los paréntesis especificamos los argumentos que dicha función necesita, (en este caso ninguno), luego se usa esta expresión (`=>`) llamada “fat arrow” y posteriormente el código de la función (en este caso es de una sola línea, con lo cual no se necesita indicar el `return` de la función. En caso de que quisiera incluir varias líneas , se debe usar las llaves de apertura y cierre `{"",""}` y al final añadir el `return`. ).

## TEMA 3: TYPESCRIPT

### Arrow Functions

- Veamos un ejemplo
  - Typescript

```
1 let getName = () => 'John Smith';
2 console.log(`The name is ` + getName());
```

- ES5

```
1 var getName = function () { return 'John Smith';
2 console.log("The name is " + getName());
3
```

## TEMA 3: TYPESCRIPT

### Arrow Functions

- Las ventajas de usar arrow functions son:
  - Sintaxis más concisa.
  - Te dan una visión más predecible de a dónde apunta la variable “**this**”:
    - Dependiendo de cómo se llame a una función , la variable **this** apunta al objeto actual (donde está corriendo el código en ese momento) o hace referencia al objeto global. Con el uso de “Arrow functions” ya siempre sabremos donde apunta **this**.

## TEMA 3: TYPESCRIPT

### Modificadores de acceso

- **Public:** accesible incluso fuera de la clase.
- **Private:** sólo dentro de la clase.
- **Protected:** sólo dentro de la clase y sus subclases.

```
1 class Person {  
2  
3     constructor(public firstName: string,  
4                 public lastName: string, public age: number, private _ssn: string) {  
5     }  
6 }  
7  
8 var p = new Person("John", "Smith", 29, "123-90-4567");  
9 console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

## TEMA 3: TYPESCRIPT

### Herencia

- El concepto de herencia en TypeScript es muy similar al que utilizamos en Java o C#.
- Básicamente hacemos uso de la palabra “**extends**” para especificar que una clase es “subclase” de otra clase.

```
1 class Person {  
2  
3     constructor(public firstName: string,  
4                 public lastName: string, public age: number,  
5                 private _ssn: string) { _;  
6     }  
7 }  
8  
9 class Employee extends Person{
```

## TEMA 3: TYPESCRIPT

### Herencia

- Actualmente, TypeScript no tiene ninguna sintaxis especial para expresar herencia múltiple. En su lugar se puede utilizar la estrategia **MIXING**.
- La estrategia **mixing** es básicamente que dejas que tu clase principal implemente las clases que deseas, escribes la implementación ficticia simple para sus interfaces y luego tienes una función especial que lee tus clases y sobrescribe las propiedades en tu clase principal.
- Como se mencionó anteriormente, (ES5) no soporta clases , y por ello tampoco usa la herencia como se conoce en (POO) . En JavaScript sería “Herencia Prototipo”.
- En Javascript no extiendes clases, extiendes objetos.

## TEMA 3: TYPESCRIPT

### Anotación genérica

- Se sabe que en (POO) , las clases y/o los constructores,(etc) pueden trabajar con múltiples tipos, Tomemos como ejemplo un array.
- En un array podemos meter objetos de varios tipos. La **anotación genérica** permite especificar qué tipo de objetos vamos a permitir que se introduzcan en el Array.
- Ejemplo:

```
1 class Person {  
2     name: string;  
3 }  
4  
5 class Employee extends Person{  
6     department: number;  
7 }  
8  
9 class Animal {  
10    breed: string;  
11 }  
12  
13 var workers: Array<Person> = [];  
14  
15 workers[0] = new Person();  
16 workers[1] = new Employee();  
17 workers[2] = new Animal();  
18
```

Compile time error

## TEMA 3: TYPESCRIPT

- Interfaces. En TypeScript , las interfaces se pueden utilizar para dos propósitos:

- Definir un tipo.

```
interface IPerson {  
    firstName: string;  
    lastName: string;  
    age: number;  
    ssn?: string;  
}
```

- el símbolo “?” en la propiedad “ssn” indica que dicha propiedad es **opcional**. Es decir, te permite crear instancias de esa clase sin la necesidad de especificar la propiedad “ssn”.

- Veamos un ejemplo

```
class Person {  
    constructor(public config: IPerson) {  
    }  
}  
  
var aPerson: IPerson = {  
    firstName: "John",  
    lastName: "Smith",  
    age: 29  
}  
  
var p = new Person(aPerson);  
console.log("Last name: " + p.config.lastName );
```

## TEMA 3: TYPESCRIPT

- Interfaces. En TypeScript , las interfaces se pueden utilizar para dos propósitos:
- Variación de implementaciones de una clase: En vez de crear dos métodos separados para cada clase , utilizaremos una interfaz

```
interface IPayable{  
  
    increasePay(percent: number): void  
}  
  
class Employee implements IPayable{  
  
    increasePay(percent: number): void {  
        // increase salary  
    }  
}  
  
class Contractor implements IPayable{  
  
    increasePay(percent: number): void {  
        // increase hourly rate  
    }  
}  
  
var workers: Array<IPayable> = [];  
workers[0] = new Employee();  
workers[1] = new Contractor();  
  
workers.forEach(worker => worker.increasePay(30));
```

## TEMA 3: TYPESCRIPT

### Desestructuración de objetos

- Imaginemos que estamos recibiendo un objeto con unas 20 propiedades, pero solo necesitamos un par de ellas. La desestructuración de objetos nos sirve para recoger directamente las únicas propiedades que deseamos fácilmente.

```
1  function getStock(){
2
3      return {
4          symbol: "IBM",
5          price: 100.00,
6          open: 99.5,
7          volume:100000
8      };
9 }
10 let {symbol, price} = getStock();
11 console.log(`The price of ${symbol} is ${price}`);
12
13
```



## TEMA 3: TYPESCRIPT

### El compilador de Typescript

- Si queremos correr, compilar código TypeScript de manera local necesitaremos instalar el compilador de TypeScript. (Ya lo tenemos instalado al haber instalado Angular con NPM)
- Formas de compilar TypeScript:
  - tsc --t ES5 main.ts // compila main.ts en main.js con la sintaxis de ECMAScript 5.
  - Crear un fichero “tsconfig.json” y dentro especificar las opciones de compilación que se desean aplicar

```
{  
  "compilerOptions": {  
    "target": "ES5",  
    "module": "commonjs",  
    "experimentalDecorators": true  
  }  
}
```

## TEMA 3: TYPESCRIPT

### El compilador de Typescript

- ¿Que módulo deberíamos utilizar?, en JavaScript hay varios sistemas de módulos que tienen diferente sintaxis, pero siempre que trabajemos con **NodeJS** usaremos el módulo “**commonjs**”
- **experimentalDecorators: true**
  - Se necesita esta opción para manejar correctamente los decoradores que dan soporte a anotaciones (como las de Java)
- Si te encuentras en la ruta donde se ubica el archivo “tsconfig.ts” sólo tendrás que ejecutar el comando “tsc” para compilar, y te generará el/los archivo/s .js en esa misma ruta.
- Para ver todas las opciones de compilación, puedes visitar la siguiente URL:
  - <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

## TEMA 3: TYPESCRIPT

### Ficheros de definición

- Como ya sabemos, podemos declarar y utilizar “tipos” en TypeScript, pero ¿y si queremos utilizar una librería externa de JavaScript? . Nosotros querríamos ser capaces de poder utilizar todos los beneficios que nos da TypeScript (tipos, avisos de error de compilación,etc) . ¿Sería posible importar una librería externa de JavaScript en mi proyecto y seguir pudiendo disponer de los beneficios de TypeScript?, la respuesta son los ficheros de definición
- **Los ficheros de Definición** (Type definition files) contienen declaraciones de tipos para las librerías de JavaScript.
- Los ficheros (\*.d.ts) ayudan a los IDE con “Type-Ahead”.
- El analizador estático de TypeScript usa los ficheros (\*.d.ts) para reportar errores.
- Puedes encontrar los (Type definition files) en la siguiente URL:
  - <https://www.npmjs.com/~types>

## TEMA 3: TYPESCRIPT

### Ficheros de definición

- Una forma de instalar un (Type definition) file en nuestro proyecto sería de esta forma:
  - `npm i @types/loadsh --save`
- **--save** : sirve para que incluya el type definition en el “package.json” y así posteriormente lo instalará siempre como una dependencia.
- **--save -dev** : lo incluye como dependencia únicamente en la sección de dependencias de “desarrollo” y no en “producción”.

## TEMA 4: COMPONENTES

Veremos las siguientes cuestiones

- Elementos de un componente
- Como declarar un componente
- Ejemplos y ejercicios
- Inyección de dependencias

## TEMA 4: COMPONENTES

Ya hemos visto un ejemplo de componente al inicio del curso veamos los detalles

- La interfaz de un componente es declarada en la plantilla o template. Observamos el selector donde pintaremos el componente que se llamará app-root

```
@Component({
  selector: 'app-root',
  template: '<h1>Hello World!</h1>'
})
```

- Si el HTML es muy largo podemos usar templateUrl para especificar el la URL en un fichero separado.
- De manera similar tenemos también styles o stylesURL. Si nos fijamos hay corchetes, es decir, podemos poner varios.

```
@Component({
  selector: 'app-root',
  stylesURL: '[app.component.css]',
  template: '<h1>Hello World!</h1>'
})
```

## TEMA 4: COMPONENTES

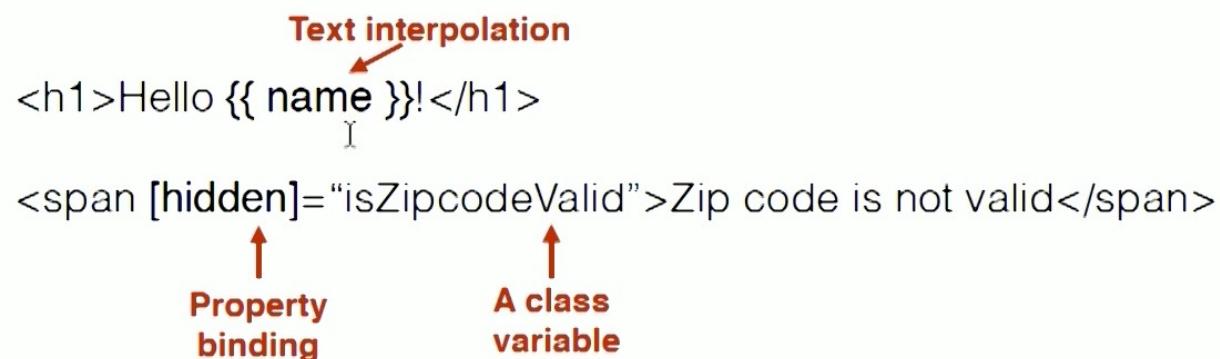
- Si no queremos generar un fichero pero queremos mejorar la visibilidad podemos usar ` para indicar que va todo junto

```
@Component({
  selector: 'auction-home-page',
  ...
  styleUrls: ['home.component.css'],
  template: `←
    <div class="row carousel-holder">
      <div class="col-md-12">
        <auction-carousel></auction-carousel>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <div class="form-group">
          <input placeholder="Filter products by title" type="text">
        </div>
      </div>
    </div>
  `→
})`
```

## TEMA 4: COMPONENTES

Nuestro componente querrá enlazar datos en la vista con nuestro controlador para eso tenemos el data binding. Data binding nos permite lo siguiente

- Sincronizar los datos de la vista y el componente.
- Cambiar la vista automáticamente cuando cambia un valor
- Cambiar los datos en los componentes si cambian en la UI
- Los elementos UI pueden ser enlazados a propiedades de nuestro componente
- Los eventos pueden ser enlazados a funciones o expresiones
- Un ejemplo



The diagram illustrates two examples of Angular data binding:

- Text interpolation:** An arrow points from the placeholder {{ name }} in the `<h1>Hello {{ name }}!</h1>` code snippet to the text "Text interpolation".
- Property binding:** An arrow points from the `[hidden]` attribute in the `<span [hidden]="isZipcodeValid">Zip code is not valid</span>` code snippet to the text "Property binding".
- A class variable:** An arrow points from the `isZipcodeValid` variable in the `<span [hidden]="isZipcodeValid">Zip code is not valid</span>` code snippet to the text "A class variable".

## TEMA 4: COMPONENTES

### Data binding

- Para enlazar eventos podemos hacer los siguiente o crear eventos propios

```
<button (click)="placeBid()">Place Bid</button>
  ↑
  DOM
  event
  ↓
<input (input)="onInputEvent()" />
  ↗

  Custom
  event
  ↓
<price-quoter (lastPrice)="priceQuoteHandler($event)">
</price-quoter>
```

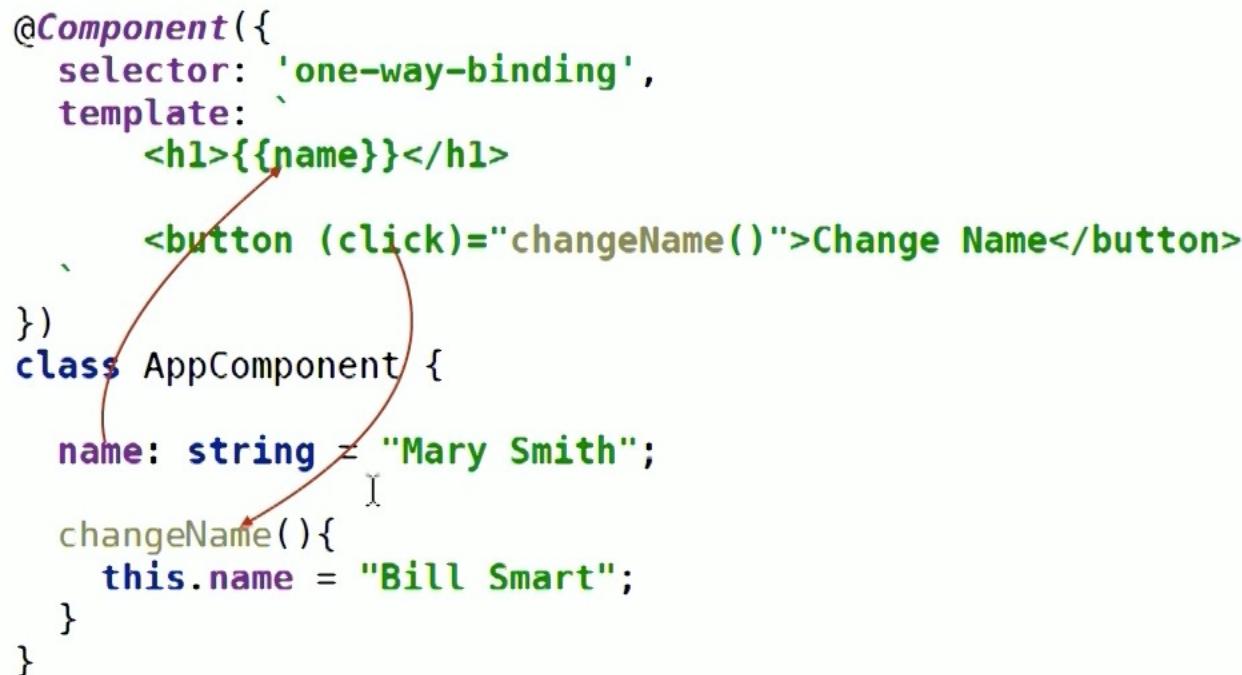
- Ejercicio: Vamos a modificar el root component que hemos creado para cambiar el título de una página

## TEMA 4: COMPONENTES

### Data binding

- Solución

```
@Component({
  selector: 'one-way-binding',
  template: `
    <h1>{{name}}</h1>
    <button (click)="changeName()">Change Name</button>
  `
})
class AppComponent {
  name: string = "Mary Smith";
  changeName(){
    this.name = "Bill Smart";
  }
}
```



The diagram illustrates the data binding process. A red arrow points from the double curly braces {{name}} in the template to the name variable in the component class. Another red arrow points from the click event in the button to the changeName() method, indicating how the UI triggers a state change.

## TEMA 4: COMPONENTES

### Two way binding

- Es posible sincronizar en los dos sentidos. Veamos como

```
@Component({
  selector: 'two-way-binding',
  template: `<input type='text' placeholder= "Enter shipping address"
    → [(ngModel)] = "shippingAddress"/>
    <button (click)="shippingAddress='123 Main Street'">Set Default Address</button>
    <p>The shipping address is {{shippingAddress}}</p>
`})
class AppComponent {
  shippingAddress: string;
}
```

## TEMA 4: COMPONENTES

### Two way binding

- Si ponemos eso en nuestro código no funcionará. Para ello debemos habilitar el soporte para two way binding importando forms
- En la definición del módulo
  - import { FormsModule } from '@angular/forms';
- En la sección de imports del mismo módulo
  - imports: [ BrowserModule, FormsModule ],...
- Ejercicio: Probemos a generar un pequeño formulario con nombre y apellidos. Crearemos un botón que al cliclarlo muestre un mensaje de bienvenida con el nombre y apellidos juntos

## TEMA 4: COMPONENTES

Veamos la inyección de dependencias

- La inyección de dependencias ofrece un control de acceso a los servicios
- No hay necesidad de crear servicios con new
- Angular crea e inyecta los servicios en nuestros componentes
- El concepto es igual a la inyección de dependencias de otros frameworks, como por ejemplo Spring
- A la hora de probar nuestra aplicación la inyección nos podría permitir cambiar un servicio remoto en un componente por uno que sea un mock para hacer nuestras pruebas sin tener que cambiar código de nuestra aplicación.

## TEMA 4: COMPONENTES

Veamos la inyección de dependencias

- Angular inyecta valores en componentes a través de los constructores
- Cada componente tiene sus propio inyector
- Hay que especificar un proveedor para que Angular sepa que inyectar

```
@NgModule ({  
    ...  
    providers: [{provide: ProductService, useClass: MockProductService}]  
})
```



- Aquí estamos diciendo que podemos inyectar de la clase MockProductService un token (o referencia) ProductService
- Normalmente se suele asociar un provider a una instancia única de un servicio

## TEMA 4: COMPONENTES

Veamos la inyección de dependencias

- Si el token y el tipo son iguales se puede usar una notación más simple
  - providers:[ProductService]
- Si se incluyen en NgModule estará disponible para todo la aplicación.
- Si se incluye en un @Component estará disponible para ese componente y sus hijos.
- Para usarlo lo haremos a través del constructor

```
constructor(productService: ProductService) {  
    productService.doSomething();  
}
```

- Aquí ya vemos que podemos usar directamente el objeto.
- Angular se encarga de resolver automáticamente de donde viene esa inyección sino la declaramos

## TEMA 4: COMPONENTES

Veamos la inyección de dependencias

- Para cambiar la inyección solo tenemos que cambiarla en el provider y dejamos el token igual
- Para especificar que un elemento es injectable hacemos la declaración así

```
import {Injectable} from "@angular/core";  
  
→ @Injectable()  
export class ProductService {  
  getProducts(): Product {  
    // An HTTP request can go here  
    return new Product( 0, "iPhone 7", 249.99, "The latest iPhone, 7-inch screen");  
  }  
}
```

```
export class Product {  
  constructor(  
    public id: number,  
    public title: string,  
    public price: number,  
    public description: string) {  
  }  
}
```

## TEMA 4: COMPONENTES

Veamos la inyección de dependencias

- Y ahora como usarlo

```
import {Component} from '@angular/core';
import {ProductService, Product} from "./product.service";

@Component({
  selector: 'di-product-page',
  template: `<div>
    <h1>Product Details</h1>
    <h2>Title: {{product.title}}</h2>
    <h2>Description: {{product.description}}</h2>
    <h2>Price: \${{product.price}}</h2>
  </div>`,
  providers: [ProductService]
})

export class ProductComponent {
  product: Product;

  constructor( productService: ProductService) {
    this.product = productService.getProduct();
  }
}
```

Injection

## TEMA 4: COMPONENTES

Veamos como crear componentes y subcomponentes

- Para crear componentes podemos hacerlo de dos formas
  - A través de la Angular CLI
  - Creando a mano los componentes
- Dependiendo de si queremos reutilizar código de un modulo a otro podemos optar por una opción u otra.
- Es posible exportar nuestro módulo para reutilizarlo en otras aplicaciones. Incluso podemos subirlo al repositorio de npm para publicarlo (de manera parecida a maven).

## TEMA 4: COMPONENTES

Veamos como crear componentes con Angular CLI

- Para generar un componente tenemos que usar el siguiente comando
  - *ng generate component NOMBRE-COMPONENTE*
  - También vale *ng g component NOMBRE-COMPONENTE*
- Esto genera una estructura de carpetas con nuestro componente
- Podemos indicar en que carpeta queremos que genere el componente
  - *ng g component feature/new-cmp*

## TEMA 4: COMPONENTES

Veamos como crear componentes con Angular CLI

- También podemos generar pipes, directivas, etc...

Scaffold	Usage
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Guard	<code>ng g guard my-new-guard</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-module</code>

## TEMA 4: COMPONENTES

Veamos como crear componentes a mano

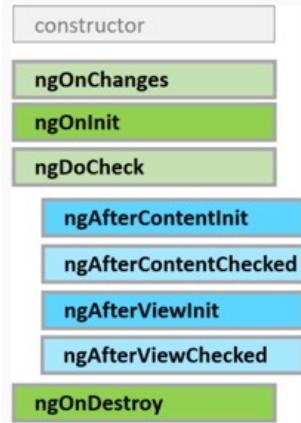
- Es posible que algún momento nos pueda interesar crear un componente a mano:
  - Para copiarlo de otra aplicación
  - Porque sea muy similar a otro componente de la aplicación y justifique el generar uno nuevo
  - Por gusto
- Tendremos que tener varias cosas en cuenta
  - Dependiendo del tipo de componente debemos generar una estructura u otra de clases y dependencias
  - Tenemos que recordar añadirlo en algún punto de nuestra aplicación, por ejemplo en nuestro módulo root o en algún sub-módulo
- No se recomienda crearlos a mano a no ser que estemos muy seguros de lo que hacemos.



## TEMA 4: COMPONENTES

### Ciclo de vida de los componentes

- Los componentes cambian de estado a lo largo de su ciclo de vida. Desde su creación con el constructor a su destrucción. Veamos a continuación el ciclo de vida de un componente.
- Para utilizar los ciclos de vida idealmente implementaremos la interfaz específica y su método como en el ejemplo aportado.
- Cada ciclo de vida aporta un hook o enlace para cambios de estado.



```
export class PeekABoo implements OnInit {
  constructor(private logger: LoggerService) { }

  // implement OnInit's `ngOnInit` method
  ngOnInit() { this.logIt('OnInit'); }

  logIt(msg: string) {
    this.logger.log(`#${nextId++} ${msg}`);
  }
}
```

## TEMA 4: COMPONENTES

### Ciclo de vida de los componentes

ngOnChanges()	Se llama antes de ngOnInit() y cuando uno o mas datos enlazados cambia.
ngOnInit()	Se llama una sola vez al inicializarse la directiva/componente después de que angular haya mostrado los datos y haya inicializado las propiedades.
ngDoCheck()	Detecta y actúa sobre cambios que Angular no puede detectar por si mismo. Se ejecuta en cada cambio inmediatamente después ngOnChanges() y ngOnInit().
ngAfterContentInit()	Responde después de que Angular proyecte contenido en la vista del componente. Se llama después del primer ngDoCheck(). Es un hook solo para componentes
ngAfterContentChecked()	Responde después de que Angular chequeé el contenido proyectado en el componente . Se llama después de ngAfterContentInit() y en después de cada ngDoCheck(). Es un hook solo para componentes
ngAfterViewInit()	Responde después de que Angular inicialice las vistas del componente y los componentes hijos. Se llama una vez después de ngAfterContentChecked(). Es un hook solo para componentes
ngAfterViewChecked()	Responde después de que Angular chequeé la vista del componente y de sus componentes hijos. Se llama después de ngAfterViewInit() y en cada ngAfterContentChecked(). Es un hook solo para componentes
ngOnDestroy	Limpieza antes de que Angular destruya la directiva/componente. Ideal para quitar Observables y gestores de eventos para evitar problemas de memoria. Se llama justo antes de que Angular destruya la directiva/componente.



## TEMA 5: DIRECTIVAS

Veremos

- Tipos de directivas
- Ejemplos y ejercicios

## TEMA 5: DIRECTIVAS

### Tipos de directivas

- Angular tiene tres tipos de directivas
  - Directivas con template o plantilla: Son las que hemos visto hasta ahora y aportan contenido a nuestra aplicación
  - Directivas de atributos: Cambian el comportamiento o apariencia de un elemento, componente u otra directiva
  - Directivas estructurales: Cambian el DOM añadiendo o quitando elementos

## TEMA 5: DIRECTIVAS

### Directivas de atributos

- Nos permiten cambiar el comportamiento o apariencia de un elemento, componente o directiva.
- Se crean con *ng g directive NOMBRE-DIRECTIVA*
- Nuestro componente tendrá un @Directive donde tendremos un selector para referenciarlo

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

## TEMA 5: DIRECTIVAS

### Directivas de atributos

- Para referenciarlo en el código tendremos que usarlo en una etiqueta
  - <p appHighlight>Highlight me!</p>
- Es posible que nuestro atributo responda a eventos iniciados por el usuarios. Para ello podemos usar los HostListener.
  - import { [Directive](#), [ElementRef](#), [HostListener](#) } from '@angular/core'; // para poder utilizarlos
  - [@HostListener](#)('mouseenter') onMouseEnter() {  
this.highlight('yellow'); } // ejemplo de uso

## TEMA 5: DIRECTIVAS

### Directivas de atributos

- Para pasar parámetros a los atributos podemos utilizar @Input para hacer el data binding.
- Para poder utilizarlo importamos Input
  - import { Directive, ElementRef, HostListener, Input } from '@angular/core';
- Declaramos un parámetro en la directiva que será el que utilicemos en la UI
  - @Input() highlightColor: string;
- Un ejemplo de uso

```
<h1>My First Attribute Directive</h1>

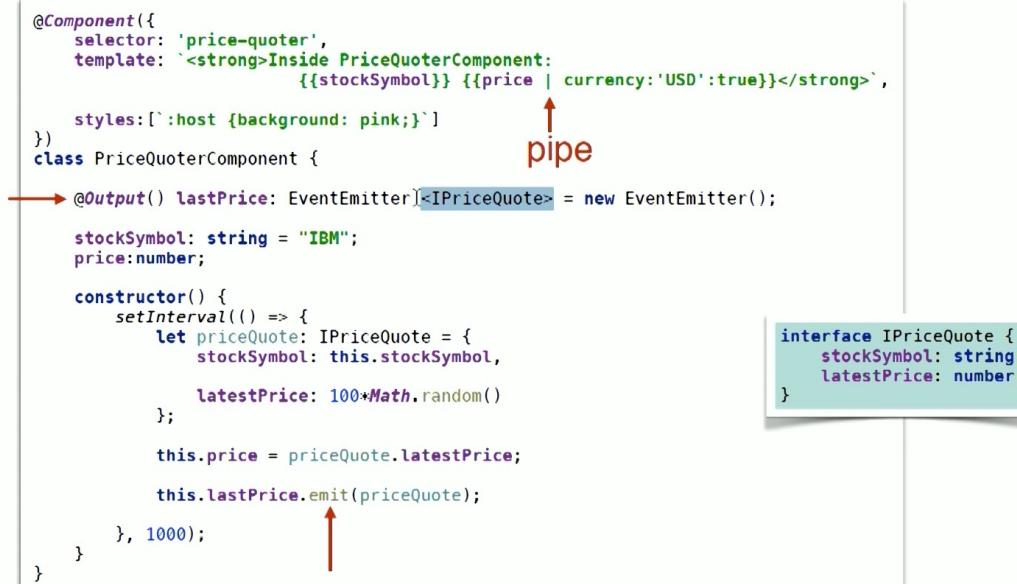
<h4>Pick a highlight color</h4>
<div>
  <input type="radio" name="colors" (click)="color='lightgreen'">Green
  <input type="radio" name="colors" (click)="color='yellow'">Yellow
  <input type="radio" name="colors" (click)="color='cyan'">Cyan
</div>
<p [appHighlight]="color">Highlight me!</p>
```

## TEMA 5: DIRECTIVAS

### Directivas de atributos

- Al igual que @Input podemos usar @Output para enviar respuestas a nuestros atributos
- En un componente podemos querer emitir un evento con EventEmmiter.

```
@Component({
  selector: 'price-quoter',
  template: `<strong>Inside PriceQuoterComponent:
    {{stockSymbol}} {{[price | currency:'USD']:true}}</strong>`,
  styles:[`:host {background: pink;}`]
})
class PriceQuoterComponent {
  @Output() lastPrice: EventEmitter<IPriceQuote> = new EventEmitter();
  stockSymbol: string = "IBM";
  price:number;
  constructor() {
    setInterval(() => {
      let priceQuote: IPriceQuote = {
        stockSymbol: this.stockSymbol,
        latestPrice: 100*Math.random()
      };
      this.price = priceQuote.latestPrice;
      this.lastPrice.emit(priceQuote);
    }, 1000);
  }
}
```



```
interface IPriceQuote {
  stockSymbol: string;
  latestPrice: number;
}
```

## TEMA 5: DIRECTIVAS

### Directivas de atributos

- Para recogerlo en un módulo que lo use (del que el modulo que hace el output no sabe nada) lo utilizamos de la siguiente manera

```
@Component({
  selector: 'app',
  template: `
    <price-quoter (lastPrice)="priceQuoteHandler($event)"></price-quoter><br>
    AppComponent received: {{stockSymbol}} {{price | currency:'USD':true}}
  `})
class AppComponent {

  stockSymbol: string;
  price:number;

  priceQuoteHandler(event:IPriceQuote) {
    this.stockSymbol = event.stockSymbol;
    this.price = event.latestPrice;
  }
}
```

## TEMA 5: DIRECTIVAS

### Directivas estructurales

- Cambian el DOM añadiendo o quitando elementos
- Son fáciles de reconocer en el código ya que llevan un asterisco
  - `<div *ngIf="hero" >{{hero.name}}</div>`

## TEMA 5: DIRECTIVAS

### Principales directivas estructurales

- `ngIf`
  - `ngIf` para true o false y undefined (que se considera como falso)
  - `<div *ngIf="persona" >{{persona.name}}</div>`
- `ngFor` para iterar sobre un array o lista
  - `<div *ngFor="let persona of personas">{{persona.name }}</div>`
  - En este caso es un div sencillo que lista los nombres de un objeto
  - `<app-persona-detail *ngFor="let persona of personas" [persona]="persona"></app-persona-detail>`
  - En este caso tenemos un array personas. Creamos una variable persona (let persona) que será la que usaremos en nuestro componente app-persona-detail
  - `ngFor` dispone de un índice que podemos utilizar
    - `<div *ngFor="let persona of personas; let i=index">{{i + 1}} - {{personas.name}}</div>`



## TEMA 5: DIRECTIVAS

### Principales directivas estructurales

- ngFor con trackBy
  - Cuando las listas son muy grandes puede interesarnos no tener que vigilar todos los elementos de un objeto.
  - TrackBy nos permite identificar un elemento para detectar si ha habido cambios que deban ser aplicados en el DOM. Por ejemplo la id de una persona.
  - Para ello debemos identificarlo en nuestro componente
    - trackByPersona(index: number, persona: Persona): number {  
    return persona.id; }
  - Para usarlo en la plantilla
    - <div \*ngFor="let persona of personas; trackBy: trackByPersonas">

## TEMA 5: DIRECTIVAS

### Principales directivas estructurales

- **ngSwitch**
  - Es muy sencillo, veamos un ejemplo.

```
<div [ngSwitch]="currentHero.emotion">  
  <app-happy-hero *ngSwitchCase="'happy'" [hero]="currentHero"></app-happy-hero>  
  <app-sad-hero *ngSwitchCase="'sad'" [hero]="currentHero"></app-sad-hero>  
  <app-confused-hero *ngSwitchCase="'confused'" [hero]="currentHero"></app-confused-hero>  
  <app-unknown-hero *ngSwitchDefault [hero]="currentHero"></app-unknown-hero>  
</div>
```

- A diferencia del switch de Java, en Angular no tenemos que especificar un break

## TEMA 5: DIRECTIVAS

### Principales directivas estructurales

- NgTemplate es un contendor de HTML que podemos mostrar en combinación con ngIf. Se usa cuando no queremos mostrar el contenido cuando no se cumple la condición. Se usa por ejemplo para las imágenes de carga
  - <div class="lessons-list" \*ngIf="lessons else loading"> ... </div> <ng-template #loading> <div>Loading...</div> </ng-template>
- Aquí lo vemos asociado a un atributo loading. Veremos los atributos más adelante

## TEMA 5: DIRECTIVAS

### Principales directivas estructurales

- NgContainer nos permite envolver elementos sin necesitar de una etiqueta HTML, por ejemplo si estamos añadiendo texto dinámicamente.

```
<p>
    I turned the corner
    <ng-container *ngIf="hero">
        and saw {{hero.name}}. I waved
    </ng-container>
    and continued on my way.
</p>
```

## TEMA 6: ROUTING

Veremos

- Estrategias para localizar componentes
- Configurar routes
- Pasar datos a los routes
- Routes hijos
- Protección de routes

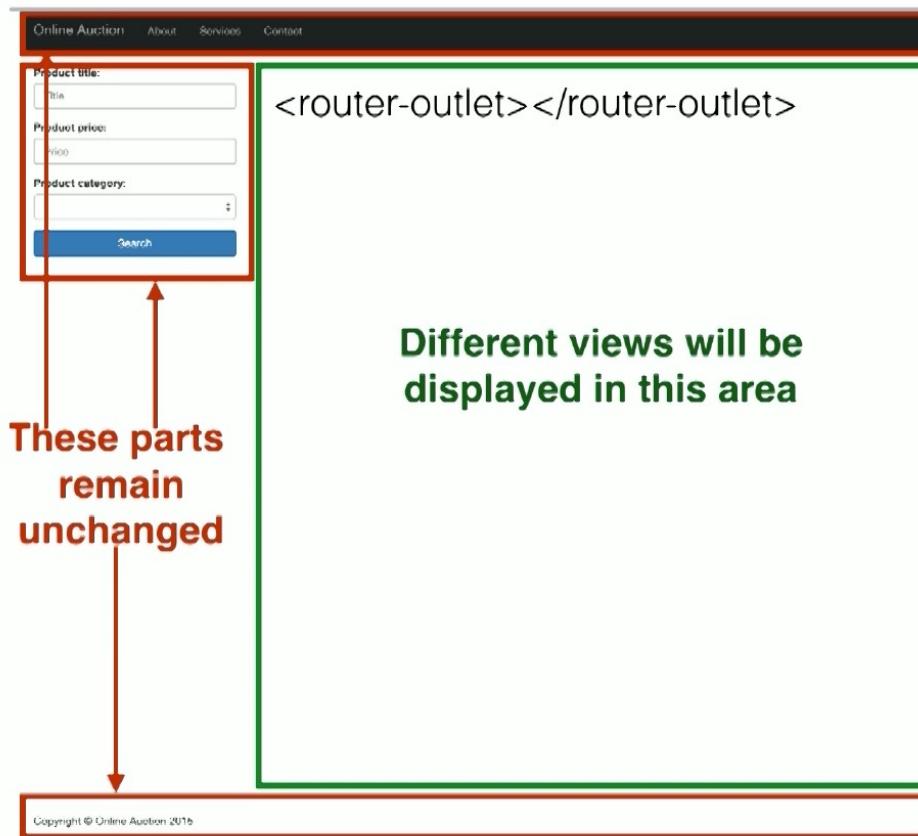
## TEMA 6: ROUTING

Que es una SPA (Single page app)

- Una SPA no refresca la página entera para mostrar diferentes vistas
- Una SPA es una colección de vistas, por ejemplo Home, detalle de producto, carrito, etc...
- Un componente router nos permite navegar de una vista a otra dentro de nuestra SPA.
- Un buen ejemplo puede ser gmail. Al visualizar nuestro inbox, un correo o responder técnicamente estamos en la misma página.

## TEMA 6: ROUTING

Si recordamos nuestro ejemplo inicial teníamos una vista donde podíamos mostrar diferentes vistas

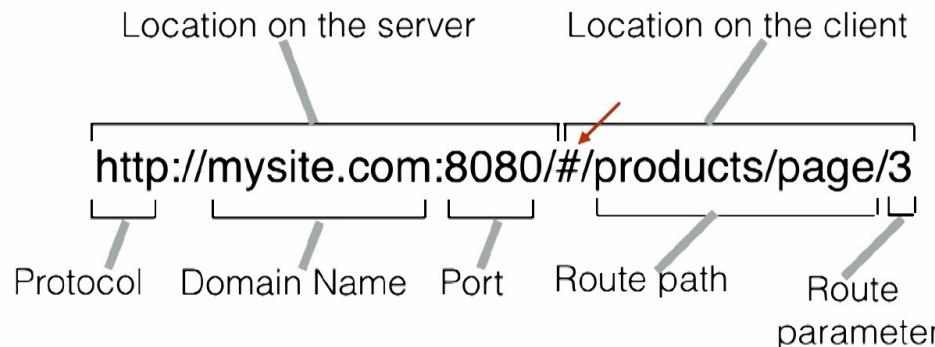


## TEMA 6: ROUTING

Veamos las estrategias de localización para los routes

- En Angular podemos mostrarla con el símbolo #.

- <https://mail.google.com/mail/u/0/#inbox>
- Para añadirlo podemos ver como en  
<https://angular.io/api/common/HashLocationStrategy>



```
@NgModule({
  ...
  providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],
  bootstrap: [ ApplicationComponent ]
})
export class AppModule { }
```

## TEMA 6: ROUTING

Veamos las estrategias de localización para los routes

- También podemos mostrarla con el path, pero solo en navegadores que soporte HTML5.
- Es el que hay por defecto. Si quisiéramos que fuera un suffix específico tendríamos que indicarlo.

http://mysite.com:8080/products/page/3

↑  
no hash

```
@NgModule({  
    ...  
    providers: [{provide: APP_BASE_HREF, useValue: '/'},  
    bootstrap: [ApplicationComponent]  
])  
export class AppModule { }
```

## TEMA 6: ROUTING

Veamos como se configuran los routes

- Tenemos que importar el componente y declarar los paths
- Después exportamos el routes para poder usarlo

The file app.routing.ts:

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home';
import { ProductDetailComponent } from './product';

const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'product', component: ProductDetailComponent}
];

export const routing = RouterModule.forRoot(routes); // routes config for the root module
```



## TEMA 6: ROUTING

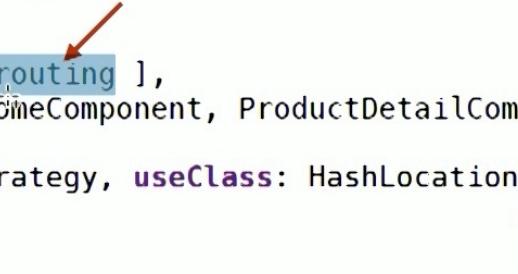
Veamos como se configuran los routes

- En nuestro modulo root hacemos la importación para poder utilizarla
- Para webapps es necesario ponerla

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './components/app.component';
import { HomeComponent} from "./components/home";
import {ProductDetailComponent} from "./components/product";
import {LocationStrategy, HashLocationStrategy} from '@angular/common';

import {routing} from './components/app.routing';

@NgModule({
  imports:      [ BrowserModule, routing ],
  declarations: [ AppComponent, HomeComponent, ProductDetailComponent ],
  providers: [{provide: LocationStrategy, useClass: HashLocationStrategy}],
  bootstrap:   [ AppComponent ]
})
class AppModule { }
```



## TEMA 6: ROUTING

Veamos como se configuran los routes

- Veamos como se usan en el UI

```
import {Component} from '@angular/core';
@Component({
  selector: 'app',
  template: `
    <a [routerLink]="/">Home</a>
    <a [routerLink]="/product">Product Details</a>

    <router-outlet></router-outlet>
  `
})
export class AppComponent {}
```

- Para hacer la navegación por código podemos usar navigate() o navigateByUrl()

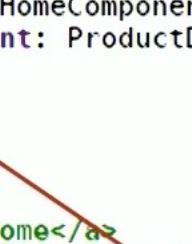
## TEMA 6: ROUTING

Veamos pasar datos en la navegación con routes

- Normalmente querremos pasar datos al navegar de un componente a otro.

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'product/:id', component: ProductDetailComponentParam}
];

@Component({
  selector: 'app',
  template:
    <a [routerLink]="/">Home</a>
    <a [routerLink]="/product", 1234">Product Details</a>
    <router-outlet></router-outlet>
  ,
})
class AppComponent {}
```



## TEMA 6: ROUTING

Veamos pasar datos en la navegación con routes

- Para recibir los parámetros tenemos que definirlo en nuestro componente. Para ello usamos ActivatedRoute que inyectamos en nuestro componente.
- Para obtener los parámetros usamos snapshot.paramMap.get(...)
- También podemos usar paramMap.subscribe para obtenerlo pero lo veremos más adelante.

```
import {Component} from '@angular/core';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'product',
  template: `<h1 class="product">Details for product {{productID}}</h1>`,
  styles: ['.product {background: cyan}']
})
export class ProductDetailComponentParam {
  productID: string;

  constructor(route: ActivatedRoute) {
    this.productID = route.snapshot.paramMap.get('id');
  }
}
```

## TEMA 6: ROUTING

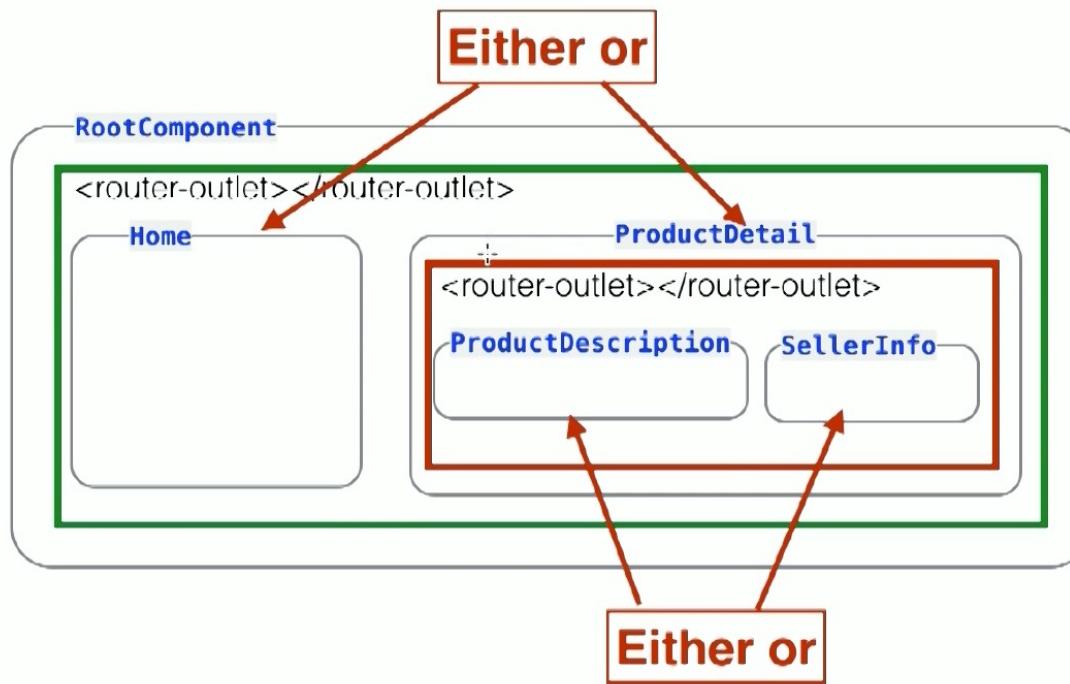
Veamos un ejercicio

- Vamos a crear un modulo root donde definiremos la navegación de dos componentes, home y product detail
- Montaremos los componentes con la misma navegación que en los ejemplos, creando una zona de routing donde mostraremos dos productos con una id diferente en el menú de navegación.

## TEMA 6: ROUTING

Veamos como hacer routing extendido con child routes

- Un componente puede tener su propio routes



## TEMA 6: ROUTING

Veamos como hacer routing extendido con child routes

- ¿Cómo lo configuramos?
- La ruta si accedieramos a la información de seller sería
  - /#/product/1234/seller/5678

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'product/:id', component: ProductDetailComponent,
   → children: [
      {path: '', component: ProductDescriptionComponent},
      {path: 'seller/:id', component: SellerInfoComponent}
    ]}
];

@Component({
  selector: 'app',
  template:
    <a [routerLink]="/">Home</a>
    <a [routerLink]="/product, 1234">Product Details</a>
    <router-outlet></router-outlet>

})
class AppComponent {}
```

## TEMA 6: ROUTING

Veamos como proteger nuestros routes

- Angular viene con varios interfaces para proteger nuestros routes
  - CanActivate: Para mediar en la navegación a una ruta y dejar que el usuario haga la acción. Por ejemplo para verificar si el usuario está logado.
  - CanActivateChild: para mediar en una navegación a un child route de manera similar a la anterior
  - CanDeactivate: para mediar en la navegación para parar la navegación. Por ejemplo cuando queremos validar que el usuario quiere hacer realmente la acción (un borrado de datos sensible)
  - Resolve: Para recoger datos antes de la activación de un módulo y así asegurarnos que los datos están listos.
  - CanLoad: Para mediar en una navegación de un módulo cargado asíncronamente.

## TEMA 6: ROUTING

Veamos como proteger nuestros routes

- Un ejemplo de canActivate y canDeactivate

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'login', component: LoginComponent},
  {path: 'product', component: ProductDetailComponent,
    canActivate:[LoginGuard],
    canDeactivate:[UnsavedChangesGuard]}
];

export const routing = RouterModule.forRoot(routes);
```

## TEMA 6: ROUTING

Veamos como proteger nuestros routes

- Si vemos el código de LoginGuard vemos que implementamos CanActivate que solo tiene el método canActivate.
- Vemos también que inyectamos en el constructor el router para usar la navegación de manera programática

```
export class LoginGuard implements CanActivate{
    constructor(private router: Router){}
    → canActivate() {
        return this.checkIfLoggedIn();
    }
    private checkIfLoggedIn(): boolean{
        // A call to the actual login service would go here
        // For now we'll just randomly return true or false
        let loggedIn:boolean = Math.random() <0.5;
        if(!loggedIn){
            alert("You're not logged in and will be redirected to Login page");
            this.router.navigate(["/login"]);
        }
        return loggedIn;
    }
}
```

## TEMA 6: ROUTING

Veamos como proteger nuestros routes

- Veamos canDeactivate
- Nuestro componente tiene una propiedad name. Dirty significa que no se han guardado los datos.

```
export class UnsavedChangesGuard implements CanDeactivate<ProductDetailComponent>{  
  constructor(private _router:Router){}  
  
  canDeactivate(component: ProductDetailComponent){  
  
    if (component.name.dirty) {  
      return window.confirm("You have unsaved changes. Still want to leave?");  
    } else {  
      return true;  
    }  
  }  
}
```

```
@Component({  
  selector: 'product',  
  template: `<h1 class="product">Product Detail Component</h1>  
           <input placeholder="Enter your name"  
                 type="text" [formControl]="name">`,  
  styles: ['.product {background: cyan}']  
})  
export class ProductDetailComponent{  
  
  name: FormControl = new FormControl();  
}
```

## TEMA 7: SERVICIOS

Veremos

- Sobre los servicios
- Ejercicios

## TEMA 7: SERVICIOS

### Servicios

- Los servicios nos permiten generar módulos que serán reutilizados en nuestra aplicación en diferentes componentes.
- Por ejemplo recuperar datos de la base de datos o un servicio web.
- Creamos un servicio con Angular CLI
  - ng g service MI-SERVICIO
  - Cuando lo crea no lo importa directamente, por lo que debemos importarlo en el módulo o módulos donde lo usemos.
  - También en los providers

## TEMA 7: SERVICIOS

### Servicios

- Veamos un ejemplo de un servicio que carga una lista de coches

```
export class DataService {  
  
    constructor() {}  
  
    cars = [  
        'Ford', 'Chevrolet', 'Buick'  
    ];  
  
    myData() {  
        return 'This is my data, man!';  
    }  
  
}
```

## TEMA 7: SERVICIOS

### Servicios

- Podríamos querer cargar esta lista al iniciar la aplicación

```
export class AppComponent {  
  
    constructor(private dataService:DataService) {  
  
    }  
  
    someProperty:string = '';  
  
    ngOnInit() {  
        console.log(this.dataService.cars);  
  
        this.someProperty = this.dataService.myData();  
    }  
  
}
```

## TEMA 7: SERVICIOS

Algunos ejemplos de módulos de servicios útiles

- Una clase con utilidades para una aplicación. Por ejemplo cálculos financieros
- Una clase que nos permita un singleton de acceso al local storage para guardar en la memoria del navegador información útil
- Una clase que permita acceder a métodos de una API remota.

## TEMA 7: SERVICIOS

### Ejercicio

- Vamos a crear un servicio que modele una clase persona. Le daremos id, nombre, apellidos y teléfono.
- Crearemos un servicio que gestione dos objetos en memoria. Un array de empleados y un usuario.
  - `localStorage.setItem('whatever', 'something');`
- Crearemos métodos en nuestro servicio para añadir empleados, borrarlos y modificarlos, también para el usuario.
- Para los más adelantados generaremos los componentes UI para tratarlos.

## TEMA 8: FORMULARIOS

En este tema veremos como utilizar los formularios en Angular

- Veremos las plantillas específicas para formularios
- Veremos como hacer formularios reactivos

## TEMA 8: FORMULARIOS

Gracias a Angular podemos tener cobertura a las siguientes funcionalidades

- Obtener valores de uno o varios formularios
- Validar de manera individual o grupal formularios
- Mostrar mensajes de error
- Detectar cambios en formularios
- Gestión de eventos
- Creación dinámica de elementos de un formulario.

## TEMA 8: FORMULARIOS

### Formularios en Angular

- Los formularios en Angular se gestionan a través de un objeto que contiene los valores y el estado del formulario y sus controles
- Este modelo se puede crear implícita o explícitamente.
- Se crean a través de las clases FormControl, FormGroup y FormArray

## TEMA 8: FORMULARIOS

### Formularios en Angular

- Cuando dejamos a Angular definir a través de plantillas con directivas se denomina gestionado por plantilla o template-driven. En este caso el modelo del formulario se crea implícitamente.
- Cuando definimos explícitamente el modelo del formulario en Typescript se llama Reactive.

## TEMA 8: FORMULARIOS

### Formularios en Angular

- Para usar los formularios debemos importarlos en el package.json, por ejemplo: “`@angular/forms": "^4.3.0"`
- Debemos implementar FormsModule si queremos usar formularios en @NgModule de manera implícita.
- Si queremos usar formularios reactive debemos importar ReactiveFormsModule

## TEMA 8: FORMULARIOS

Veamos un ejemplo de template

- Importamos FormsModule en @NgModule
- Usamos las directivas NgForm, NgModel, NgGroup

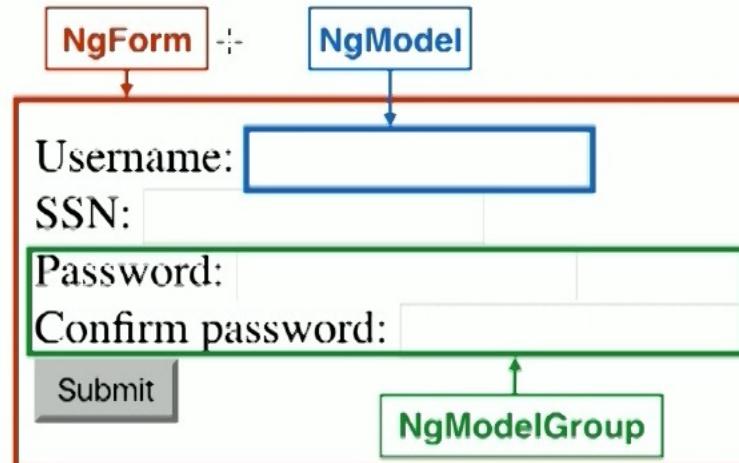
```
template: `↓          ↓
  <form #f="ngForm" (ngSubmit)="onSubmit(f.value)">
    <div>Username:      <input type="text"    name="username" ngModel></div>
    <div>SSN:           <input type="text"    name="ssn"       ngModel></div>
    <div>Password:       <input type="password" name="password" ngModel></div>
    <button type="submit">Submit</button>
  </form>
```

- En este ejemplo definimos una variable f para usarla después.
- Con ngSubmit podemos enlazar con un método en nuestro módulo.
- Debemos asignar un nombre para los input de nuestro formulario.

## TEMA 8: FORMULARIOS

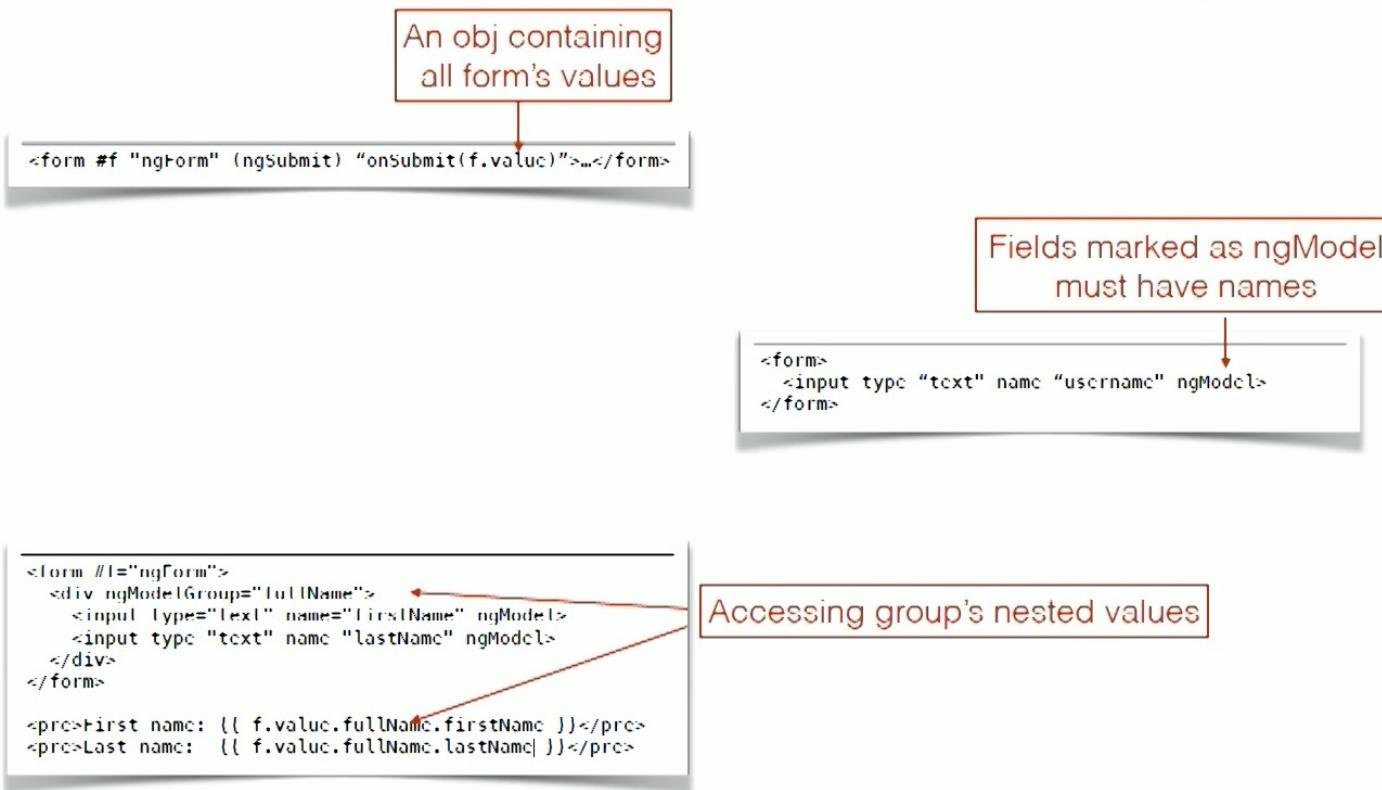
Veamos un ejemplo de template

- `ngForm` representa un formulario
- `ngModel` representa un campo de nuestro formulario
- `ngModelGroup` permite agrupar campos juntos.



## TEMA 8: FORMULARIOS

### Veamos un ejemplo de template



## TEMA 8: FORMULARIOS

### Controladores Reactive

- Antes vimos los tipos de clases que soportaban formularios, veámoslas en detalle:
  - FormControl: Representa un único elemento de un formulario
  - FormGroup: Representa una parte del formulario y es una colección de FormControls
  - FormArray: Es similar a FormGroup. Es útil para crear campos de manera dinámica.

## TEMA 8: FORMULARIOS

### Controladores Reactive

- Todos descienden de AbstractControl lo que nos da muchas propiedades útiles.

```
class AbstractControl {  
    constructor(validator: ValidatorFn, asyncValidator: AsyncValidatorFn)  
    validator : ValidatorFn  
    asyncValidator : AsyncValidatorFn  
    value : any  
    status : string  
    valid : boolean  
    invalid : boolean  
    pending : boolean  
    disabled : boolean  
    enabled : boolean  
    errors : {[key: string]: any}  
    pristine : boolean  
    dirty : boolean  
    touched : boolean  
    untouched : boolean  
    valueChanges : Observable<any>  
    statusChanges : Observable<any>  
    setValidators(newValidator: ValidatorFn|ValidatorFn[]): void  
    ...
```

## TEMA 8: FORMULARIOS

### Controladores Reactive

- Veamos algunos fragmentos reactive (creamos nosotros los formularios)

#### formGroup

```
<form [FormGroup] = "myFormModel"></form>
...
@Component( ... )
class FormComponent {
    myFormModel: FormGroup = new FormGroup({ ... });
}
```

#### formGroupName

```
<form [FormGroup] = "myFormModel">
    <div formGroupName="employmentDates">...</div>
</form>
...
@Component( ... )
class FormComponent {
    myFormModel: FormGroup = new FormGroup({
        employmentDates: new FormGroup({
            fromDate: new FormControl(),
            toDate: new FormControl()
        })
    })
}
```

#### FormControlName

```
<form [FormGroup] = "myFormModel" (ngSubmit) = "onSubmit()">
    <div>Username: <input type="text" formControlName="username"></div>
    <div>SSN: <input type="text" formControlName="ssn"></div>
</form>
...
class AppComponent {
    myFormModel: FormGroup;
    constructor() {
        this.myFormModel = new FormGroup({
            username: new FormControl(),
            ssn: new FormControl()
        });
    }
}
```

#### formControl, no <form> tag

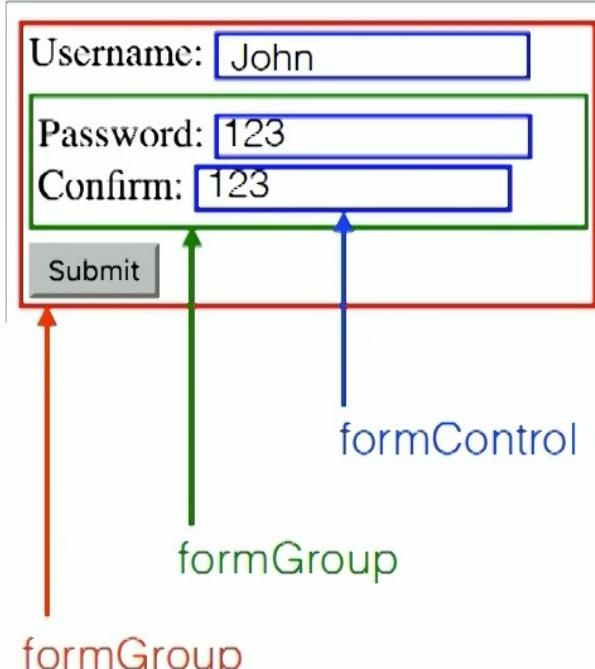
```
<input type="text" [FormControl] = "weatherControl">
...
class WeatherComponent {
    weatherControl: FormControl = new FormControl();
    constructor() {
        this.weatherControl.valueChanges
            .debounceTime(500)
            .switchMap(city > this.getWeather(city))
            .subscribe(weather > console.log(weather));
    }
}
```

## TEMA 8: FORMULARIOS

### Controladores Reactive

- Veamos como se organizan los elementos

Rendered page:



A sample form value

```
▼ Object {username: "John", passwordGroup: Object}
  ▼ passwordGroup: Object
    password: "123"
    pconfirm: "123"
    ► __proto__: Object
    username: "John"
    ► __proto__: Object
```

## TEMA 8: FORMULARIOS

### Controladores Reactive

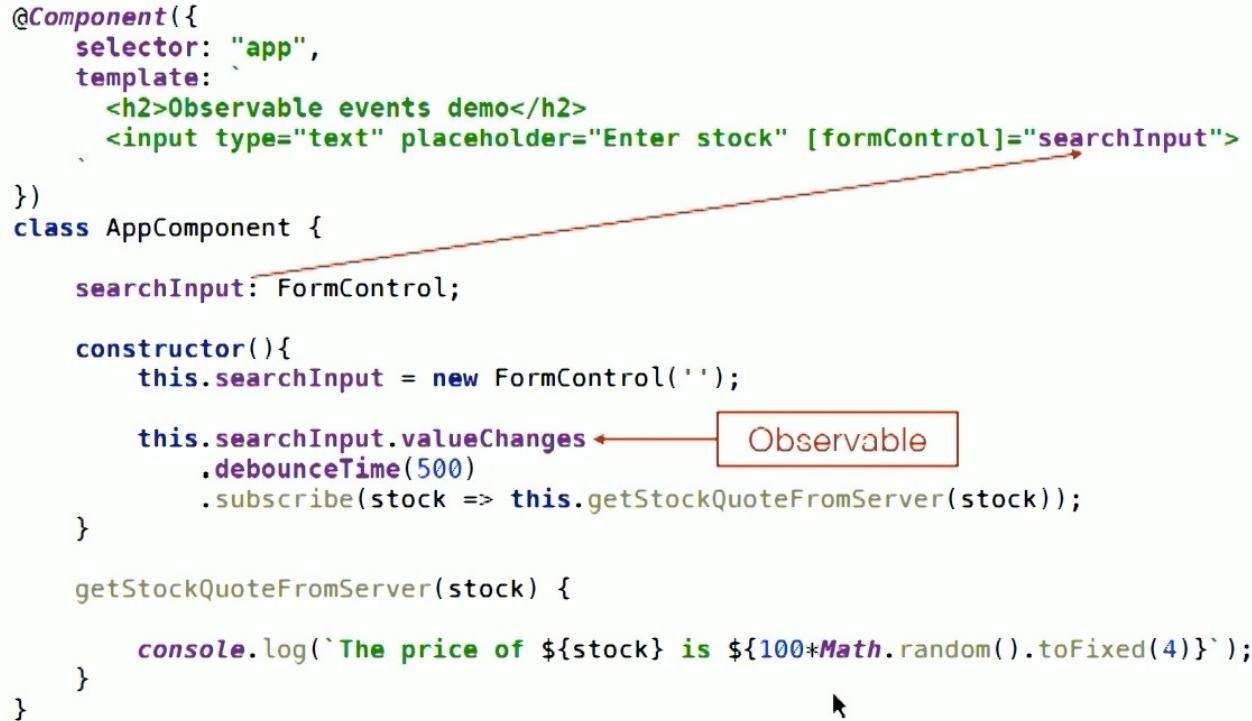
- Veamos otro ejemplo. En este caso un buscador

```
@Component({
  selector: "app",
  template:
    <h2>Observable events demo</h2>
    <input type="text" placeholder="Enter stock" [formControl]="searchInput">
})
class AppComponent {
  searchInput: FormControl;

  constructor(){
    this.searchInput = new FormControl('');

    this.searchInput.valueChanges
      .debounceTime(500)
      .subscribe(stock => this.getStockQuoteFromServer(stock));
  }

  getStockQuoteFromServer(stock) {
    console.log(`The price of ${stock} is ${100*Math.random().toFixed(4)}`);
  }
}
```



## TEMA 8: FORMULARIOS

### Controladores Reactive

- Paso a paso para la creación de controladores Reactive

- 1. Importar ReactiveFormsModule en nuestra app.module

```
import { ReactiveFormsModule }
        from '@angular/forms';

...
@NgModule({
  imports: [ BrowserModule, ReactiveFormsModule ],
```

- 2. Crear un modelo que representa el formulario instanciando las clases del modelo

```
@Component({...})
class AppComponent {
  myFormModel: FormGroup = new FormGroup([
    emails: new FormArray([
      new FormControl()
    ])
});
```

- 3. Crear una plantilla usando directivas react

```
<form [formGroup]="myFormModel"
       (ngSubmit)="onSubmit()">
  ...
</form>
```

- 4. Usar una instancia de FormGroup para acceder a los valores del formulario

```
onSubmit() {
  console.log(this.myFormModel.value);
}
```

## TEMA 9: PIPES

En este tema veremos

- Para que sirven los componentes pipe
- Como se crean los componentes pipe
- Ejemplos y ejercicios

## TEMA 9: PIPES

### Pipes

- Los pipes sirven para hacer transformaciones sobre datos y mostrarlos en la UI.
- Son muy útiles para formateos de fecha, moneda, etc relacionado con la internacionalización.
- También pueden ser muy útiles para transformar objetos.
- Veamos un ejemplo
  - template: `<p>The hero's birthday is {{ birthday | date }}</p>`

## TEMA 9: PIPES

### Pipes Definidos en Angular

- Angular nos da pipes ya creados para poder usarlos directamente. Veamos los más importantes
  - [DatePipe](#) para tratamiento de fechas
  - [UpperCasePipe](#) y [LowerCasePipe](#) para tratamiento de cadenas
  - [CurrencyPipe](#) para tratamiento de monedas
  - [PercentPipe](#) para tratamiento de números.
- Los pipes admiten parámetros. Por ejemplo
  - {{ birthday | date:"MM/dd/yy" }}
- También podemos enlazarlos a expresiones
  - {{ birthday | date:format }} // donde format puede ser esta llamada
  - get format() { return this.toggle ? 'shortDate' : 'fullDate'; }
- Los pipes se pueden encadenar uno detrás de otros
  - {{ birthday | date | uppercase}}

## TEMA 9: PIPES

### Como se crean los pipes en Angular

- Para crear nuestro pipe debemos generar un componente de tipo pipe como vimos en el tema 4.
- Cuando creamos el componente veremos que implementa la interfaz PipeTransform
- Esta interfaz tiene un método transform que es el que debemos implementar.
- Cuando lo generamos la notación es la siguiente
  - transform(value: any, args?: any): any {
  - Esto nos indica que puede tener cualquier valor (que podemos modificar)
  - Podemos enviar o no argumentos (el simbolo de interrogación indica que no es obligatorio)
  - Y devuelve cualquier cosa. Todo esto lo podemos cambiar para tipos específicos.

## TEMA 9: PIPES

### Como se crean los pipes en Angular

- Por ejemplo, una transformación que elimine todos los undefined de una cadena y la ponga a mayúsculas

```
transform(value: string, args?: any): string {
  var buffer: string;
  buffer = value.toString();
  console.log(buffer.indexOf(undefined));
  while (buffer.indexOf(undefined)>-1){
    buffer = buffer.replace(undefined, "");
    console.log(buffer);
  }

  return buffer.toUpperCase();
}
```

## TEMA 10: HTTP

En este tema veremos

- Trabajaremos con Angular Http
- Peticiones de datos
- Enviar datos al servidor
- Interceptores y escuchadores
- Consideraciones de seguridad
- Ejemplos y ejercicios

## TEMA 10: HTTP

### Instalando Http

- Para usar Http necesitamos importarlo en nuestro módulo
  - `import {HttpClientModule} from '@angular/common/http';`
- También necesitamos añadirlo a imports
  - `imports: [BrowserModule, HttpClientModule, ]`
- Ya podemos injectarlo en nuestros componentes declarandolo en nuestro constructor.
- Idealmente separaremos en servicios Http para poder exportarlos y reutilizarlos. Es interesante incluso separarlos por tipología. Por ejemplo, un servicio para gestionar usuarios, otro para conectarnos a una API remota (si es que no existe ya), etc...

## TEMA 10: HTTP

### Peticiones de datos

- Angular da una integración muy buena con JSON y RESTful.
- Si quisiéramos utilizar otra tecnología podemos utilizar un módulo de terceros.
- Un ejemplo de datos en JSON
  - { "results": [ "Item 1", "Item 2", ] }
- Para obtener datos usamos el objeto http con get. Para leer los datos usamos el objeto data de esta manera en nuestro componente.
  - import { HttpClient } from '@angular/common/http';
  - constructor(private http: HttpClient)...
  - http.get('/api/items').subscribe(data => { this.results = data['results']; })
- Ejemplo. Cread un archivo en assets que tenga solo json con la estructura de persona e invocadlo con get. Mapeadlo a un objeto persona y mostradlo por pantalla.

## TEMA 10: HTTP

### Peticiones de datos

- Hay bastante más para el get. Podemos obtener más información de un servicio con observe

```
http
  .get<MyJsonData>('/data.json', {observe: 'response'})
  .subscribe(resp => {
    // Here, resp is of type HttpResponse<MyJsonData>.
    // You can inspect its headers:
    console.log(resp.headers.get('X-Custom-Header'));
    // And access the body directly, which is typed as MyJsonData as requested.
    console.log(resp.body.someField);
  });
}
```

- O indicar un tipo específico de datos
  - http.get('/textfile.txt', {responseType: 'text'})

## TEMA 10: HTTP

### Peticiones de datos

- Es importante recoger los errores, para ello tenemos el objeto err.

```
http
  .get<ItemsResponse>('/api/items')
  .subscribe(
    // Successful responses call the first callback.
    data => {...},
    // Errors will call this callback instead:
    err => {
      console.log('Something went wrong!');
    }
  );
}
```

- Podemos reintentar por defecto un número de veces (por ejemplo un servicio externo que sepamos que falla a menudo con retry)
  - `http .get<ItemsResponse>('/api/items').retry(3).subscribe(...);`

## TEMA 10: HTTP

### Peticiones de datos

- A veces no estamos seguros de cuanto tiempo tardará el servicio web en responder, bien porque pone en cola nuestra petición, bien porque debe hacer un proceso largo para dar con el resultado. Para ello podemos hacer uso de las promesas.
- Llamada normal
  - `http.get('/api/users').subscribe(data => { console.log(data); });`
- Llamada con promesa
  - `http.get('/api/users').toPromise().then(data => { console.log(data)});`

## TEMA 10: HTTP

### Peticiones de datos

- Si quisieramos recoger el error exacto tenemos que estudiar el status.

```
1. http
2.   .get<ItemsResponse>('/api/items')
3.   .subscribe(
4.     data => {...},
5.     (err: HttpErrorResponse) => {
6.       if (err.error instanceof Error) {
7.         // A client-side or network error occurred. Handle it
8.         // accordingly.
9.         console.log('An error occurred:', err.error.message);
10.      } else {
11.        // The backend returned an unsuccessful response code.
12.        // The response body may contain clues as to what went wrong,
13.        console.log(`Backend returned code ${err.status}, body was:
14.          ${
15.        );
```

## TEMA 10: HTTP

### Envío de datos

- De manera similar a get tenemos también post para enviar datos.

```
const body = {name: 'Brad'};  
  
http  
  .post('/api/developers/add', body)  
  // See below - subscribe() is still necessary when using post().  
  .subscribe(...);
```

- Es importante añadir el subscribe o no se realizarán llamadas.

## TEMA 10: HTTP

### Envío de datos

- Si quisiéramos añadir un token de autenticación podemos hacerlo a través de la clase HttpHeaders

```
http
  .post('/api/items/add', body, {
    headers: new HttpHeaders().set('Authorization', 'my-auth-token'),
  })
  .subscribe();
```

## TEMA 10: HTTP

### Envío de datos

- Al igual que Headers también podemos enviar Parámetros HTTP con la clase HttpParams

```
http
  .post('/api/items/add', body, {
    params: new HttpParams().set('id', '3'),
  })
  .subscribe();
```

## TEMA 10: HTTP

### Interceptores

- Los interceptores nos permiten interceptar las llamadas de http y ejecutar acciones antes de la llamada.
- En ocasiones puede ser interesante usar interceptores para autenticación o login.
- Para usarla tenemos que importar HttpInterceptor en nuestro componente.

## TEMA 10: HTTP

### Interceptores

- Para escribir un interceptor crearemos un servicio como este.
  - Intercept se encarga de interceptar nuestra petición y next deja seguir la ejecución.

```
import {Injectable} from '@angular/core';
import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from
  '@angular/common/http';

@Injectable()
export class NoopInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

## TEMA 10: HTTP

### Interceptores

- Una vez creado lo añadimos a nuestro módulo.
  - multi a true indica que declaramos para un array de interceptores.

```
import {NgModule} from '@angular/core';
import {HTTP_INTERCEPTORS} from '@angular/common/http';

@NgModule({
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: NoopInterceptor,
      multi: true,
    },
  ],
})
export class AppModule {}
```

## TEMA 10: HTTP

### Interceptores

- Ejemplo: Reescribir todas las peticiones de http a https

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
{
    // This is a duplicate. It is exactly the same as the original.
    const dupReq = req.clone();

    // Change the URL and replace 'http://' with 'https://'
    const secureReq = req.clone({url: req.url.replace('http://', 'https://')});
}
```

## TEMA 10: HTTP

# Interceptores

- Ejemplo: Introducir una cabecera de autenticación

```
1. import {Injectable} from '@angular/core';
2. import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from
   '@angular/common/http';
3.
4. @Injectable()
5. export class AuthInterceptor implements HttpInterceptor {
6.   constructor(private auth: AuthService) {}
7.
8.   intercept(req: HttpRequest<any>, next: HttpHandler):
   Observable<HttpEvent<any>> {
9.     // Get the auth header from the service.
10.    const authHeader = this.auth.getAuthorizationHeader();
11.    // Clone the request to add the new header.
12.    const authReq = req.clone({headers: req.headers.set('Authorization',
   authHeader)});
13.    // Pass on the cloned request instead of the original request.
14.    return next.handle(authReq);
15.  }
16. }
```

## TEMA 10: HTTP

### Consideraciones de seguridad

- Uno de los elementos recurrentes al realizar aplicaciones con Angular. Esta sección es un listado de recomendaciones no exhaustiva.
  - Usar los modelos de datos. Cuando se produce un error al intentar hacer el parseo podemos detectar ataques.
  - Uso de IP o DNS autorizados
  - Uso de servicios autorizados
  - Tomar medidas cuando se producen varios errores de autenticación (por ejemplo borrar el localStorage)
  - Si guardamos información en el navegador guardarla encriptada.
  - Políticas de passwords duras y de cambios frecuentes
  - Si permitimos guardar el login (recordar contraseña) es recomendable que no sea indefinido y obligar a introducir la contraseña.
  - Auditoría de cuentas sin uso (en servidor)
  - Auditoría de accesos y acciones

## TEMA 10: HTTP

### Integraciones de autenticación con terceros

- Un problema recurrente es la autenticación con terceros. Desde redes sociales a LDAP.
- Aunque Angular no tiene un servicio por si mismo si que se pueden usar servicios de terceros como auth0.com
- <https://auth0.com/authenticate/angular2/ldap/>

## TEMA 11: TESTING

En este tema veremos

- Test unitarios
- Test de aplicación

## TEMA 11: TESTING

### Test Unitarios

- Cuando generamos nuestros componentes generamos un archivo .spec.ts donde podemos generar tests para nuestros componentes.
- Usando npm test podemos ejecutar nuestros tests.
- Podemos visualizar los resultados en localhost:9876

## TEMA 11: TESTING

### Test Unitarios

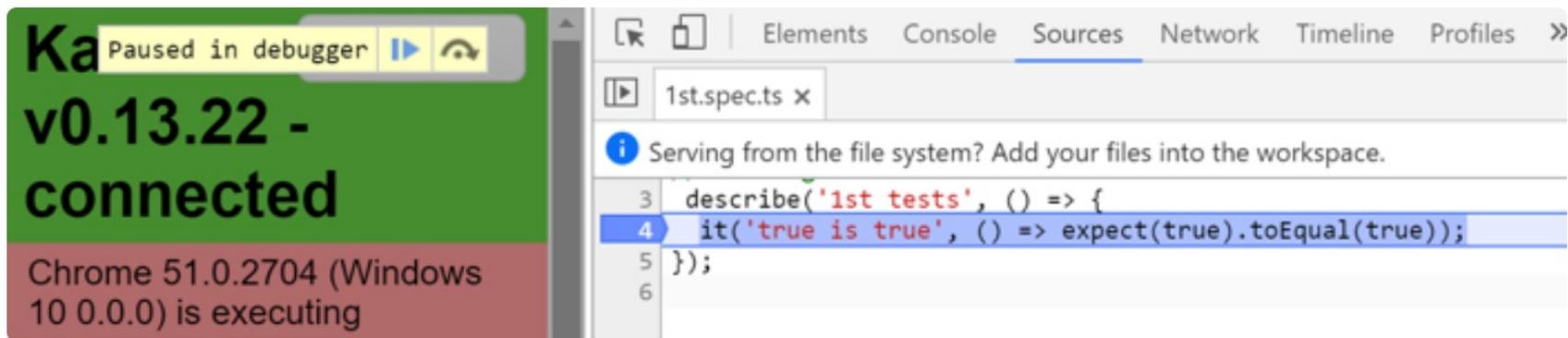
- Veamos un ejemplo

```
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';
describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));
  it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  }));
  it(`should have as title 'app'`, async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('app');
  }));
  it('should render title in a h1 tag', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.querySelector('h1').textContent).toContain('Welcome to app!');
  }));
});
```

## TEMA 11: TESTING

### Test Unitarios

- Podemos debugear la aplicación con el navegador



## ANEXO A: EJERCICIO PRÁCTICO

### Objetivo

- Vamos a trabajar con todos los elementos que hemos trabajado a lo largo del curso. Para ello vamos a utilizar todos los elementos que hemos ido viendo a lo largo del curso.
- También plantearemos paso a paso como construir una aplicación en pasos que permitan involucrar al máximo número de personas en el proyecto.
- Para trabajar la parte de HTTP vamos a trabajar con la capa de servicios ReqresIn. <https://reqres.in/> o bien con la de github o gitlab
- Para el planteamiento del ejercicio trabajaremos en grupo toda la clase.

## ANEXO A: EJERCICIO PRÁCTICO

### Paso 1: definir el alcance del proyecto

- Queremos realizar una SPA que nos permita trabajar tareas de github.
- Para ello tendremos que usar un token de usuario proporcionado por github.com
- Generaremos un CRUD para gestionar las tareas de un proyecto y un pequeño Dashboard de control con algunas estadísticas.

## ANEXO A: EJERCICIO PRÁCTICO

### Paso 2: definir la arquitectura de la aplicación

- Para realizar el proyecto debemos definir lo siguiente:
  - Arquitectura de componentes
  - Navegación
  - Distribución de servicios (web y memoria)
  - Modelos de la aplicación
  - Pruebas

## ANEXO A: EJERCICIO PRÁCTICO

### Paso 3: ejecución

- Ejecutaremos el proyecto. Podemos trabajar en grupos de 2-3 personas.

## ANEXO B: RECETAS

- I18n: Angular 7 da cobertura para la internacionalización de nuestro contenido
  - <https://angular.io/guide/i18n>