INDRA OPEN UNIVERSITY
PRESENTACION DEL CURSO

indra

ESCUELA:
TECNOLOGÍA

ANGULAR 2+

indra

ÍNDICE

ANGULAR 2.0

ESCUELA:
TECNOLOGÍA

- NOMBRE APELLIDO PROFESOR
  Javier Martín
- VER PERFIL COMPLETO:
- CONTACTO

jmartin@grupoloyal.com

Angular 7

https://angular.io/

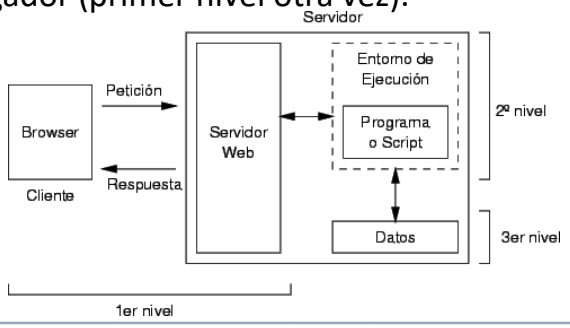
Contenidos

- Introducción a TypeScript
  - Sistema de tipos
  - Sintaxis ampliada
  - Clases, herencia, genéricos e interfeces
  - Módulos y decoradores
- Introducción a Angular 2
  - Características
  - Angulas JS vs Angular 2
  - Arquitectura
  - Elementos estructurales
- Herramientas de Desarrollo
  - Instalación de utilidades
  - Creación de una aplicación
  - Estructura de la aplicación
  - Librerías de terceros
- Módulos
  - Metadata
  - Módulo principal
  - Módulos de características
  - Módulos Angular 2 vs JavaScript
- Servicios
  - Clases como servicios
  - Dependency injection
  - Proveedores
  - Inyectores
- Componentes
  - ViewModel
  - Plantillas y estilos
  - Propiedades de entrada
  - Eventos de salida
  - Ciclo de vida
- Estilos
  - Selectores especiales
  - Encapsulación
  - Preprocesadores CSS
- Plantillas
  - Data binding
  - Marcadores
  - Directivas
  - Transformaciones (Pipes)
- Formularios
  - Vinculación bidireccional
  - Validaciones
  - Estilos visuales
- Acceso al servidor
  - Patrón Observable (RxJS).
  - HttpModule y JsonpModule
  - Servicios RESTFul
  - JSONP
- Enrutamiento y navegación
  - RouterModule
  - Definición de rutas
  - Paso de parámetros
  - Navegación

INTRODUCCIÓN A ANGULAR 2+

Arquitectura Web

- Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



Single-page application (SPA)

- Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que utiliza una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.
- En un SPA todo el códigos de HTML, JavaScript y CSS se carga de una sola vez o los recursos necesarios se cargan dinámicamente cuando lo requiera la página y se van agregando, normalmente como respuesta de los acciones del usuario.
- La página no se tiene que cargar otra vez en ningún punto del proceso, tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación.
- La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está por detrás, habitualmente utilizando AJAX o WebSocket (HTML5).

Introducción

- Angular (comúnmente llamado "Angular 2+" o "Angular 2") es un framework estructural para aplicaciones web de una sola página (SPA) siguiendo el patrón MVVM y orientación a componentes.
- Permite utilizar el HTML como lenguaje de plantillas y extender la sintaxis del HTML para expresar los componentes de la aplicación de forma declarativa, clara y sucinta .
- El enlazado de datos y la inyección de dependencias eliminan gran parte del código que de otra forma se tendría que escribir. Y todo sucede dentro del navegador, lo que lo convierte en un buen socio para cualquier tecnología de servidor.
- Angular es un framework que pretende definir la arquitectura de la aplicación.
- Define claramente la separación entre el modelo de datos, el código, la presentación y la estética.
- Estás obligado por Angular a seguir esa estructura. Ya no haces el JavaScript como tu quieres sino como Angular te manda. Y eso realmente da mucha potencia a la aplicación ya que, al seguir una arquitectura definida, Angular puede ayudarte en la aplicación y a tener todo mucho mas estructurado.
- Angular es la evolución de AngularJS aunque incompatible con su predecesor.

Novedades

- En AngularJS la escalabilidad estaba limitada por:
  - Lenguaje JavaScript
  - Precarga del código.
- Uso de TypeScript: validación, intellisense y refactoring.
- Desarrollo basado en componentes.
- Uso de Anotaciones (metadatos): declarativa frente a imperativa.
- Doble binding, se ha reconstruido según el patrón observable, las aplicaciones son hasta cinco veces más rápidas.
- La inyección de dependencias, carga dinámica de módulos.
- Incluye compilación AoT (Ahead of Time - Antes de tiempo, que es una forma de compilar todo el código JS, incluyendo plantillas de manera anticipada, restando carga de trabajo al navegador) y también lazy-loading (descarga bajo demanda, una manera de sólo descargar los recursos realmente necesarios en cada momento).
- Aparición de Angular Universal (herramienta de renderizado de Angular desde el servidor), podemos ejecutar Angular en el servidor.
- Aplicaciones híbridas, solución low cost para aplicaciones móviles.
- Está cada vez más orientado a grandes desarrollos empresariales.

Características

- Velocidad y rendimiento
  - **Generación de código**: Angular convierte sus plantillas en un código que está altamente optimizado para las máquinas virtuales de JavaScript de hoy, brindándole todos los beneficios del código escrito a mano con la productividad de un marco.
  - **Universal**: Sirva la primera vista de su aplicación en Node.js, .NET, PHP y otros servidores para renderización casi instantánea en solo HTML y CSS. También allana el camino para sitios que optimizan para SEO.
  - **División del código**: Las aplicaciones Angular se cargan rápidamente con el nuevo enrutador de componentes, que ofrece división automática de códigos para que los usuarios solo carguen el código requerido para procesar la vista que solicitan.
- Plataformas cruzadas
  - **Aplicaciones web progresivas**: Utilice las capacidades de la plataforma web moderna para ofrecer experiencias similares a las de las aplicaciones de escritorio o nativas. Instalación de alto rendimiento, fuera de línea y en cero pasos.
  - **Nativo**: Cree aplicaciones móviles nativas con estrategias de Cordova, Ionic o NativeScript.
  - **Escritorio**: Cree aplicaciones instaladas en el escritorio en Mac, Windows y Linux utilizando la misma metodología Angular que ha aprendido para la web y la capacidad de acceder a las API nativas del sistema operativo.

Características

- Productividad
  - **Plantillas**: Cree rápidamente vistas de IU con sintaxis de plantilla simple y potente.
  - **Angular CLI**: Herramientas de línea de comandos: comience a construir rápidamente, agregue componentes y pruebas, y luego implemente al instante.
  - **IDEs**: Obtenga la finalización inteligente de códigos, errores instantáneos y otros comentarios en editores e IDE populares.
- Ciclo completo de desarrollo
  - **Pruebas**: Con Karma para pruebas unitarias, puede saber si ha roto cosas cada vez que guarda. Y Protractor hace que tus pruebas de escenarios (E2E) se ejecuten más rápido y de manera estable.
  - **Animación**: Cree coreografías y líneas de tiempo de animación complejas y de alto rendimiento con muy poco código a través de la API intuitiva de Angular.
  - **Accesibilidad**: Cree aplicaciones accesibles con componentes compatibles con ARIA, guías para desarrolladores y una infraestructura de prueba incorporada.

Angular adopta SEMVER

- El sistema SEMVER (Semantic Versioning) es un conjunto de reglas para proporcionar un significado claro y definido a las versiones de los proyectos de software.
- El sistema SEMVER se compone de 3 números, siguiendo la estructura X.Y.Z, donde:
  - X se denomina Major: indica cambios rupturistas
  - Y se denomina Minor: indica cambios compatibles con la versión anterior
  - Z se denomina Patch: indica resoluciones de bugs (compatibles)
- Básicamente, cuando se arregla un bug se incrementa el patch, cuando se introduce una mejora se incrementa el minor y cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.
- De este modo cualquier desarrollador sabe qué esperar ante una actualización de su librería favorita. Si sale una actualización donde el major se ha incrementado, sabe que tendrá que ensuciarse las manos con el código para pasar su aplicación existente a la nueva versión.

Frecuencia de lanzamiento

- En general, se puede esperar el siguiente ciclo de publicación:
  - Un lanzamiento importante (major) cada 6 meses
  - 1-3 lanzamientos menores para cada lanzamiento principal
  - El lanzamiento de un parche casi todas las semanas
- Para mejorar la calidad y permitir ver lo que vendrá después, se realizan lanzamientos de versiones Beta y Release Candidatos (RC) para cada versión mayor y menor.
- Dentro de la política de desaprobación, para ayudar a garantizar que se tenga tiempo suficiente y una ruta clara de actualización, se admite cada API obsoleta hasta al menos dos versiones principales posteriores, lo que significa al menos 12 meses después de la desaprobación.
- Ruptura de la secuencia:
  - Pasa de la versión 2 → 4:
  - El componente Router ya estaba en una 3.xx

Novedades 4.0

- Actualización de TypeScript 2.1
- Uso de StrictNullChecks de TypeScript
- Sintaxis if…else dentro de los templates (*ngIf)
- Módulo de animación separado
- Integración de Angular Universal como módulo de Angular Core
- Mejoras de rendimiento gracias a FESM

```
npm install @angular/common@next @angular/compiler@next
@angular/compiler-cli@next @angular/core@next
@angular/forms@next @angular/http@next @angular/platform-
browser@next @angular/platform-browser-dynamic@next
@angular/platform-server@next @angular/router@next
@angular/animations@next --save
```

Novedades 5.0

- Actualización de TypeScript 2.4.2
- Mejoras del compilador
- API Angular Universal State Transfer y soporte DOM
- Reconstruidos los pipes de números, fechas y monedas para mejorar la internacionalización.
- Validaciones y actualizaciones de valores en `blur` o en` submit`.
- Actualizado el uso de RxJS a 5.5.2 o posterior
- Nuevos eventos del ciclo de vida del enrutador.

```
npm install @angular/animations@'^5.0.0' @angular/common@'^5.0.0'
@angular/compiler@'^5.0.0' @angular/compiler-cli@'^5.0.0'
@angular/core@'^5.0.0' @angular/forms@'^5.0.0'
@angular/http@'^5.0.0' @angular/platform-browser@'^5.0.0'
@angular/platform-browser-dynamic@'^5.0.0' @angular/platform-
server@'^5.0.0' @angular/router@'^5.0.0' typescript@2.4.2 rxjs@'^5.5.2'
--save
```


Novedades 6.0

- Lanzamiento enfocado menos en el marco subyacente, y más en la cadena de herramientas y en facilitar el movimiento rápido con Angular en el futuro, sincronizando las versiones de Angular (@angular/core, @angular/common, @angular/compiler, etc.), Angular CLI y Angular Material + CDK.
- Nuevos comandos CLI: ng update <package> y ng add <package>
- Nuevos generadores CLI: application, library, universal
- Angular Elements (Web Components)
- Angular Material + CDK Components y generadores de componentes de arranque Angular Material
- CLI v6 ahora es compatible con espacios de trabajo que contienen múltiples proyectos, como múltiples aplicaciones o bibliotecas.
- Soporte para crear y construir bibliotecas.
- Referenciado de proveedores en los servicios (Inyección de dependencias).
- Actualizada la implementación de Animaciones para que ya no necesitemos el polyfill de web animations.
- Angular ha sido actualizado para usar v.6 de RxJS.

Novedades 7.0

- Incorporación de avisos de CLI
- Mejoras Rendimiento de la aplicación
- Angular Material y el CDK:
  - Desplazamiento virtual
  - Arrastrar y soltar
  - Mejorada la Accesibilidad de selecciones
- Angular Elements ahora admite la proyección de contenido
- Actualización de TypeScript 3.1
- Angular ha sido actualizado para usar v.6.3 de RxJS.

Actualización

- El equipo de Angular a creado la Angular Update Guide para guiar a través del proceso de actualización y para conocer los cambios que se deben realizar en el código antes de comenzar el proceso de actualización, los pasos para actualizar la aplicación y la información sobre cómo prepararse para futuras versiones de Angular.

  https://angular-update-guide.firebaseapp.com/

TypeScript

TypeScript

- El lenguaje TypeScript fue ideado por Anders Hejlsberg – autor de Turbo Pascal, Delphi, C# y arquitecto principal de .NET- como un supraconjunto de JavaScript que permitiese utilizar tipos estáticos y otras características de los lenguajes avanzados como la Programación Orientada a Objetos.
- TypeScript es un lenguaje Open Source basado en JavaScript y que se integra perfectamente con otro código JavaScript, solucionando algunos de los principales problemas que tiene JavaScript:
  - Falta de tipado fuerte y estático.
  - Falta de “Syntactic Sugar” para la creación de clases
  - Falta de interfaces (aumenta el acoplamiento ya que obliga a programar hacia la implementación)
  - Módulos (parcialmente resuelto con require.js, aunque está lejos de ser perfecto)
- TypeScript es un lenguaje con una sintaxis bastante parecida a la de C# y Java, por lo que es fácil aprender TypeScript para los desarrolladores con experiencia en estos lenguajes y en JavaScript.

TypeScript

- Lo que uno programa con TypeScript, tras grabar los cambios, se convierte en JavaScript perfectamente ejecutable en todos los navegadores actuales (y antiguos), pero habremos podido:
  - abordar desarrollos de gran complejidad,
  - organizando el código fuente en módulos,
  - utilizando características de auténtica orientación a objetos, y
  - disponer de recursos de edición avanzada del código fuente (Intellisense, compleción de código, “snippets”, etc.), tanto en Visual Studio, como en otros editores populares, como Sublime Text, Eclipse, WebStorm, etc., cosa impensable hasta este momento, pero factible ahora mismo gracias a los “plug-in” de TypeScript disponibles para estos editores.

TypeScript

- Amplia la sintaxis del JavaScript con la definición y comprobación de:
  - Tipos básicos: booleans, number, string, Any y Void.
  - Clases e interfaces
  - Herencia
  - Genéricos
  - Módulos
  - Anotaciones
- Otra de las ventajas que otorga Typescript es que permite comunicarse con código JavaScript ya creado e incluso añadirle “tipos” a través de unos ficheros de definiciones .d.ts que indican los tipos que reciben y devuelven las funciones de una librería.
- Ya existen ficheros de definiciones para la mayoría de los framework mas populares (http://definitelytyped.org/).

TypeScript

- El TypeScript se traduce ("transpila": compilar un lenguaje de alto nivel a otro de alto nivel) a JavaScript cumpliendo el estándar ECMAScript totalmente compatible con las versiones existentes.
- De hecho, el "transpilador" deja elegir la versión ECMAScript del resultado, permitiendo adoptar la novedades mucho antes de que los navegadores las soporten: basta con dejar el resultado en la versión mas ampliamente difundida.
- El equipo de TypeScript provee de una herramienta de línea de comandos para hacer esta compilación, se encuentra en npm y se puede instalar con el siguiente comando:
  - npm install -g typescript
- Podemos usarlo escribiendo
  - tsc helloworld.ts
- Los diversos "plug-in" se pueden descargar en:
  - https://www.typescriptlang.org

Comandos tsc

- Compilación continua:
  - tsc –w *.ts
  - tsc --watch *.ts
- Fichero único de salida
  - tsc --outFile file.js *.ts
- Compilación del proyecto:
  -p DIRECTORY, --project DIRECTORY
- Control de directorios
  --rootDir LOCATION ← Ficheros de entrada
  --outDir LOCATION ← Ficheros de salida
- Directorio base de las rutas de los módulos:
  --baseUrl
- Creación del fichero de configuración tsconfig.json:
  --init

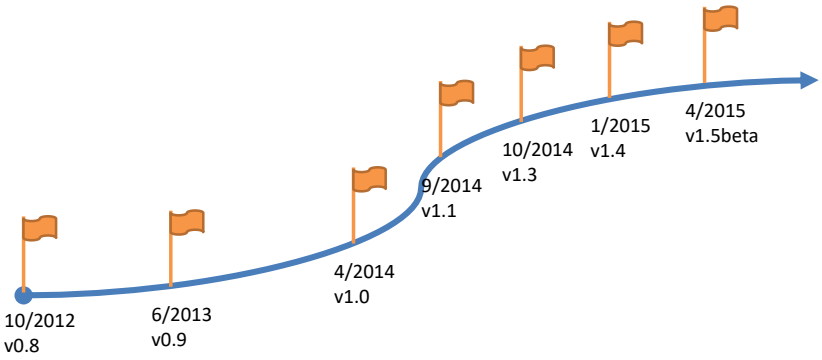
Soportado por IDE's/Utilidades

- Visual Studio 2012 … – native (+msbuild)
- Visual Studio Code
- ReSharper – included
- Sublime Text 2/3 – official plugin as a part of 1.5 release
- Online Cloud9 IDE
- Eclipse IDE
- IntelliJ IDEA
- Grunt, Maven, Gradle plugins

Eclipse

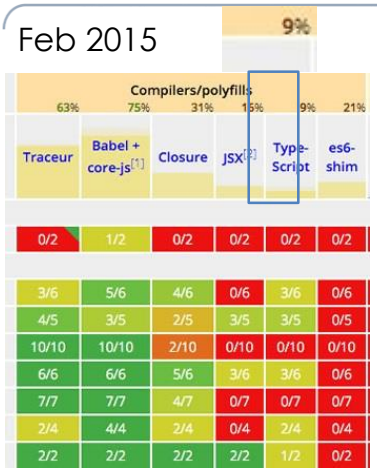
- Requiere tener instalado Node.js y TypeScript
- Abrir Eclipse
- Ir a Help → Install New Software
  - Add the update site: http://eclipse-update.palantir.com/eclipse-typescript/
  - Seleccionar e instalar TypeScript
  - Re arrancar Eclipse
- Opcionalmente, con el botón derecho en el proyecto seleccionar Configure → Enable TypeScript Builder
- En Project → Properties → Builders
  - Add new "Program" builder
  - Location: ...\tsc
  - Working Directory: ${project_loc}
  - Arguments: --source-map --outFile build.js **/*.ts
- En Project → Properties → Build Automatically

Historia

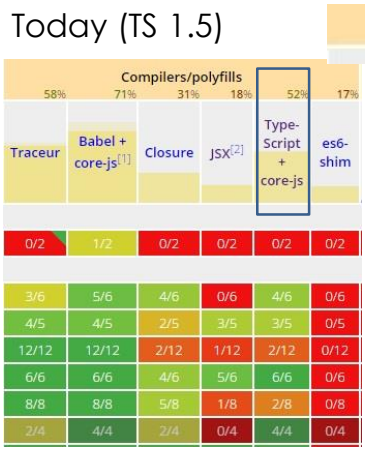


Compatibilidad con ES6

Feb 2015



Today (TS 1.5)



https://kangax.github.io/compat-table/es6/#typescript

Tipos de datos

- Boolean: var isDone: boolean = false;
- Number: var height: number = 6;
- String: var name: string = "bob";
- Enum: enum Color {Red, Green, Blue};
  var c: Color = Color.Green;
- Array: var list:number[] = [1, 2, 3];
  var list:Array<number> = [1, 2, 3];
- Any: var notSure: any = 4;
  notSure = "maybe a string instead";
  notSure = false; // okay, definitely a boolean
  var list:any[] = [1, true, "free"];
- Void: function warnUser(msg: string): void { alert(msg); }
- Null, Undefined, Never
- Object: no es number, string, boolean, symbol, null, o undefined.
- Clases e interfaces

Definiciones

- Variables globales:
  var str: string = 'Inicio';
- Variables locales:
  let x, y, z: number; // undefined
- Variables constantes:
  const LEVEL: number = 4;
- Inferencia de tipo:
  let str = 'Inicio'; // string
  let x = 1, y = 1; // number
- Tuplas:
  let tupla: [number, string] = [1, "cadena"];
  let x : number = tupla[0];
- Alias a los tipos:
  type Tupla = [number, string];
  let tupla: Tupla = [1, "cadena"];

Sintaxis ampliada (ES6)

- Cadenas (Interpolación y múltiples líneas)

```
let msg = `El precio de ${producto} es
  de ${unidades * precio} euros`;
```

- Bucle foreach:

```
for (var i of [10, 20, 30) { window.alert(i); }
```

- Destructing

```
let x = 1;
let y = 2;
[x, y] = [y, x];

let point = {x: 1, y: 2};
{x, y} = point;
```

- Auto nombrado:

```
let x = 1;
let y = 2;
let point = {x, y}; // {x: x, y: y}
```


Control de tipos

```
1  var myNumber: number = 5;
2
3  var myString = "5";
4
5  var strResult = myNumber + myString;
6
7  var object = { field: 1, field2: myString }
8
9  var err1 = object.field3;
10
11 var err2 = myNumber + object;
```

Funciones con tipo

- Tipos para los parámetros y el valor de retorno:
  function add(x: **number**, y: number): **number** { return x+y; }
- Opcionales y Valores por defecto: Se pueden definir valores por defecto a los parámetros en las funciones o marcarlos opcionales (undefined).
  function(opcional**?**: any, valor **= "foo"**) {...};
- Resto de los parámetros: Convierte una lista de parámetros en un array de Any.

```
function f (x: number, y: number, ...a): number {
   return (x + y) * a.length
}
f(1, 2, "hello", true, 7) === 9
```

- Operador de propagación: Convierte un array o cadena en una lista de parámetros.

```
var str = "foo"
var chars = [ ...str ] // [ "f", "o", "o" ]
```


Expresiones Lambda
Arrow functions

- Funciones anónimas:

```
let rslt = data.filter(item => item.value > 0);
// equivale a: var rslt = data.filter(function (item) { return item.value > 0; });
data.forEach(elem => {
   console.log(elem);
   // …
});
var fn = (num1, num2) => num1 + num2;
pairs = evens.map(v => ({ even: v, odd: v + 1 }));
```

```
1 class Greeter {
2     static greetingMessage = "Hi, ";
3
4     constructor(private personName: string) {
5     }
6     greet = () => {
7        return Greeter.greetingMessage + this.personName;
8     }
9 }
```

Tipos Unión

```
1  function f(x: number | number[]) {
2      if (typeof x === "number") {
3          //x is a number here
4          return x + 10;
5      } else {
6          //x is a number[] here
7          let sum = 0;
8
9          x.forEach((element) => {
10             sum += element;
11         });
12
13         return sum;
14     }
15 }
16
17 f(10);
18 f([10, 20]);
```


Null y Undefined (v2.1)

- En TypeScript, tanto el nulo como el sin definir tienen sus propios tipos denominados undefined y null respectivamente.
- Al igual que el vacío, no son muy útiles por su cuenta.

  ~~let u: undefined = undefined;~~

  ~~let N: null = null;~~
- Por defecto, null y undefined son subtipos de todos los demás tipos, por lo que se puede asignar null o undefined a cualquier otro tipo.
- Cuando se utiliza el flag --strictNullChecks, null y undefined sólo se pueden asignar a void y sus respectivos tipos.
  - Esto ayuda a evitar muchos errores comunes.
  - Si se desea pasar en un tipo o nulo o indefinido, se puede utilizar el tipo unión: tipo | null | undefined.

Objetos JS

```
1  function Person(age, name, surname) {
2      this.age = age;
3      this.name = name;
4      this.surname = surname;
5  }
6
7  Person.prototype.getFullName = function () {
8      return this.name + " " + this.surname;
9  }
10
11 var cadaver = new Person(60, "John", "Doe");
12
13 var boy = Person(10, "Vasya", "Utkin"); //bad
```


Clases

```
1  class Person {
2      age: number;
3      name: string;
4      surname: string;
5
6      constructor(age: number,
7          name: string,
8          surname: string) {
9          this.age = age;
10         this.name = name;
11         this.surname = surname;
12     }
13
14     getFullName() {
15         return this.name + ' ' + this.surname;
16     }
17 }
18
19 var cadaver = new Person(60, "John", "Doe");
20
21 var boy = Person(10, "Vasya", "Utkin"); //compile error
```

Constructor

```
1  class Person {
2      constructor(
3          public age: number,
4          public name: string,
5          public surname: string) {
6      }
7
8      getFullName() {
9          return this.name + ' ' + this.surname;
10     }
11 }
12
13 var cadaver = new Person(60, "John", "Doe");
14
15 var boy = Person(10, "Vasya", "Utkin"); //compile error
```


Compila a ES5

```
1  var Person = (function () {
2      function Person(age, name, surname) {
3          this.age = age;
4          this.name = name;
5          this.surname = surname;
6      }
7      Person.prototype.getFullName = function () {
8          return this.name + ' ' + this.surname;
9      };
10     return Person;
11 })();
12 var cadaver = new Person(60, "John", "Doe");
```

También a ES6

```
1  class Person {
2      constructor(age, name, surname) {
3          this.age = age;
4          this.name = name;
5          this.surname = surname;
6      }
7      getFullName() {
8          return this.name + ' ' + this.surname;
9      }
10 }
11 var cadaver = new Person(60, "John", "Doe");
```


Modificadores

- De acceso:
  - public ← *por defecto*
  - private
  - protected *(a partir de 1.3)*
- De propiedades:
  - set
  - get
- Miembros de clase:
  - static
- Clases y miembros abstractos:
  - abstract
- Atributos de solo lectura:
  - readonly (deben inicializarse en su declaración o en el constructor)

Propiedades

```
class Employee {
   private name: string;
   get Name(): string { return this.name; }
   set Name(newName: string) {
      if (this.validate('Name')) {
         this.name = newName;
         this.NameChanged();
      } else {
         // …
      }
   }
   // …
}
let employee = new Employee();
if (employee.Name === '') { employee.Name = 'Bob Smith'; }
```


Herencia

```
1  class Animal {
2      constructor(public name: string) { }
3      move(meters: number) {
4          alert(this.name + " moved " + meters + "m.");
5      }
6  }
7
8  class Snake extends Animal {
9      constructor(name: string) { super(name); }
10     move() {
11         alert("Slithering...");
12         super.move(5);
13     }
14 }
15
16 var sam = new Snake("Sammy the Python");
17
18 sam.move();
19
```

Genéricos

```
1  class Greeter<T> {
2      greeting: T;
3      constructor(message: T) {
4          this.greeting = message;
5      }
6      greet() {
7          return this.greeting;
8      }
9  }
10
11 var greeter = new Greeter<string>("Hello, world");
```


Interfaces

```
1  interface IMovable {
2      move();
3  }
4
5  class Chair implements IMovable {
6      move() {
7          console.log("Somebody moved me!");
8      }
9  }
10
11 function moveTwice(object: IMovable) {
12     object.move();
13     object.move();
14 }
15
16 moveTwice(new Chair());
```

Interfaces: Como prototipos

- Definición en línea, sin interfaces:

```
function printLabel(labelledObj: {label: string}) {
    console.log(labelledObj.label);
}
var myObj = {size: 10, label: "Size 10 Object"}; printLabel(myObj);
```

- Usando interfaces:

```
interface LabelledValue {
    label: string;
}
function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}
var myObj = {size: 10, label: "Size 10 Object"}; printLabel(myObj);
```


Interfaces: Propiedades opcionales

```
interface SquareConfig {
    color?: string;
    width?: number;
}
function createSquare(config: SquareConfig):
    {color: string; area: number} {
    var newSquare = {color: "white", area: 100};
    if (config.color) {
        newSquare.color = config.color;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}
var mySquare = createSquare({color: "black"});
```

Interfaces: Funciones y Arrays

- Prototipos de Funciones

```
interface SearchFunc {
        (source: string, subString: string): boolean;
}
var mySearch: SearchFunc;
mySearch = function(source: string, subStr: string) {
        var result = source.search(subStr);
         return result != -1;
}
```

- Prototipos de Arrays

```
interface StringArray {
        [index: number]: string;
}
var myArray: StringArray;
myArray = ["Bob", "Fred"];
```


Módulos

```
1  module Sayings {
2      export class Greeter {
3          greeting: string;
4          constructor(message: string) {
5              this.greeting = message;
6          }
7          greet() {
8              return "Hello, " + this.greeting;
9          }
10     }
11 }
12 var greeter = new Sayings.Greeter("world");
```

Módulos (ES6)

- Ficheros como módulos:
  - Solo se puede importar lo previamente exportado.
  - Es necesario importar antes de utilizar
  - El fichero en el from sin extensión y ruta relativa (./ ../) o sin ruta (NODE_MODULES)
- Exportar:

  export public class MyClass { }

  export { MY_CONST, myFunction, name as otherName }
- Importar:

  import * from './my_module';

  import * as MyModule from './my_module';

  import { MyClass, MY_CONST, myFunction as func } from './my_module';
- Pasarelas: Importar y exportar (index.ts)

  export { MyClass, MY_CONST, myFunction as func } from './my_module';

Decoradores (ES7)

```
1  class C {
2    @readonly
3    @enumerable(false)
4    method() { }
5  }
6
7  function readonly(target, key, descriptor) {
8      descriptor.writable = false;
9  }
10
11 function enumerable(value) {
12   return function (target, key, descriptor) {
13      descriptor.enumerable = value;
14   }
15 }
```

Compatibilidad con Frameworks JS

- d.ts
- Github/DefinitelyTyped

```
declare module Sayings {
    class Greeter {
        greeting: string;
        constructor(message: string);
        greet(): string;
    }
}
declare var greeter: Sayings.Greeter;
```


Angular 2.0 – Escrito en TypeScript

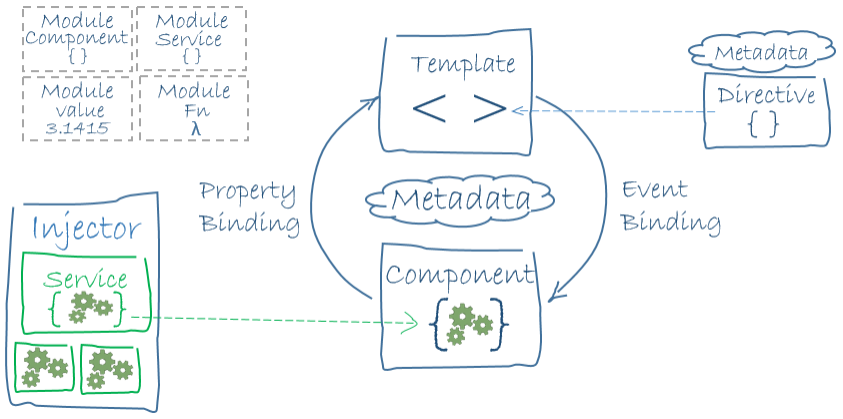
```
import {Component, View, bootstrap}
from 'angular2/angular2';

@Component({
    selector: 'my-app'
})
@View({
    template: '<h1>Hello {{name}}</h1>'
})
class MyAppComponent{
    name: string;
    constructor() {
        this.name = 'Alice'
    }
}

bootstrap(MyAppComponent);
```

ARQUITECTURA DE ANGULAR

Arquitectura de Angular



Módulo

- La aplicaciones Angular son modulares.
- Angular dispone de su propio sistema de modularidad llamado módulos Angular o NgModules .
- Cada aplicación Angular tiene al menos un módulo, el módulo raíz, convencionalmente denominado AppModule.
- Mientras que el módulo de raíz puede ser el único módulo en una aplicación pequeña, en la mayoría de las aplicaciones existirán muchos más “módulos de características”, cada uno como un bloque coherente de código dedicado a un dominio de aplicación, un flujo de trabajo o un conjunto estrechamente relacionado de capacidades.
- El módulo Angular, una clase decorada con @NgModule, es una característica fundamental de Angular.

Bibliotecas Angular

- Los contenedores Angular son una colección de módulos de JavaScript, se puede pensar en ellos como módulos de biblioteca.
- Cada nombre de la biblioteca Angular comienza con el prefijo @angular.
- Se pueden instalar con el gestor de paquetes NPM e importar partes de ellos con instrucciones import del JavaScript.
- De esta manera se está utilizando tanto los sistemas de módulos de Angular como los de JavaScript juntos.

Módulos Angular vs. Módulos JavaScript(ES6)/TypeScript

- JavaScript tiene su propio sistema de módulos para la gestión de colecciones de objetos de JavaScript.
- Es completamente diferente y sin relación con el sistema de módulos Angular.
- En JavaScript cada archivo es un módulo y todos los objetos definidos en el archivo pertenece a ese módulo.
- El módulo declara algunos objetos para ser públicos marcándolas con la palabra clave export.
- Los módulos de JavaScript usan declaraciones de importación para tener acceso a los objetos públicos (exportados) de dichos módulos.
- Se trata de dos sistemas diferentes y *complementarios* de módulos.

Metadatos

- Angular ha optado por sistema declarativo de definición de sus elementos mediante el uso de decoradores (ES7), también conocidos como anotaciones o metadatos.
- Los Decoradores son funciones que modifican las clases de JavaScript.
- Angular suministra decoradores que anotan a las clases con metadatos y le indican lo que significan dichas clases y cómo debe trabajar con ellas.

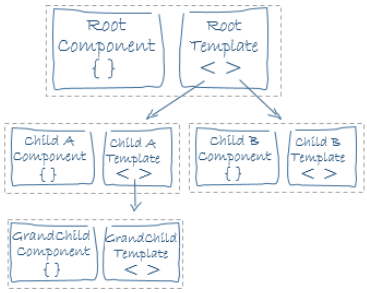
```
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Componentes

- Un componente controla una parte de la pantalla denomina vista.
- Se puede considerar que un componte es una clase con interfaz de usuario.
- Reúne en un único elemento: el código (clase que expone datos y funcionalidad), su representación (plantilla) y su estética (CSS).
- El código interactúa con la plantilla a través del enlazado dinámico.
- Una vez creado, un componente es como una nueva etiqueta HTML que puede ser utilizada en otras plantillas.
- Los componentes son elementos estancos que establecen una frontera clara y concisa de entrada/salida.
- Angular crea, actualiza y destruye los componentes según el usuario se mueve a través de la aplicación.
- Angular permite personalizar mediante el uso de enganches (hooks) lo que pasa en cada fase del ciclo de vida del componente.

Plantillas

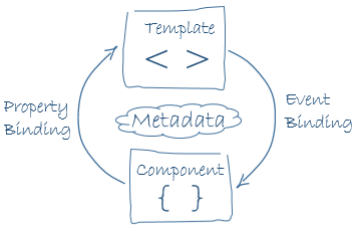
- Las plantillas contienen la representación visual de los componentes.
- Son segmentos escritos en HTML ampliado con elementos suministrados por Angular como las directivas, marcadores, pipes, …
- Cuentan con la lógica de presentación pero no permiten la codificación: los datos y la funcionalidad se los suministra la clase del componente.

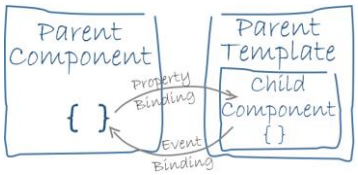


Una plantilla puede utilizar otros componentes, actuan como contenedores, dichos componentes pueden contener a su vez otros componentes. Esto crea un árbol de componentes: el modelo de composición.

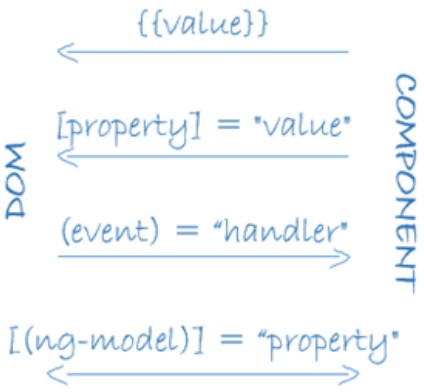
Enlace de datos

- Automatiza, de una forma declarativa, la comunicación entre los controles HTML de la plantilla y las propiedades y métodos del código del componente.
- Requieren un nuevo modelo mental declarativo frente al imperativo tradicional.
- El enlace de datos también permite la comunicación entre los componentes principales y secundarios.





Enlace de datos



- La comunicación ente Componente y Plantilla puede ser:
  - Unidireccional:
    - Datos (C→P)
      - Interpolación
      - Propiedades
    - Comandos (P→C)
      - Eventos
  - Bidireccional:
    - Datos (P←→C)
      - Directivas

Directivas

- Las directivas son marcas en los elementos del árbol DOM, en los nodos del HTML, que indican al Angular que debe asignar cierto comportamiento a dichos elementos o transformarlos según corresponda.
- Podríamos decir que las directivas nos permiten añadir comportamiento dinámico al árbol DOM, haciendo uso de las directivas propias del Angular o extender la funcionalidad hasta donde necesitemos creando las nuestras propias.
- La recomendación es que el único sitio donde se puede manipular el árbol DOM debe ser en las directivas, para que entre dentro del ciclo de vida de compilación, binding y renderización del HTML que sigue el Angular.
- Aunque un componente es técnicamente una directiva-con-plantilla, conceptualmente son dos elementos completamente diferentes.

Pipes

- El resultado de una expresión puede requerir alguna transformación antes de estar listo para mostrarla en pantalla: mostrar un número como moneda, que el texto pase a mayúsculas o filtrar una lista y ordenarla.
- Los Pipes de Angular, anteriormente denominados filtros, son simples funciones que aceptan un valor de entrada y devuelven un valor transformado. Son fáciles de aplicar dentro de expresiones de plantilla, usando el operador tubería (|).
- Son una buena opción para las pequeñas transformaciones, dado que al ser encadenables un conjunto de transformaciones pequeñas pueden permitir una transformación compleja.

Servicios

- Los servicios en Angular son una categoría muy amplia que abarca cualquier valor, función o característica que necesita una aplicación.
- Casi cualquier cosa puede ser un servicio, aunque típicamente es una clase con un propósito limitado y bien definido, que debe hacer algo específico y hacerlo bien.
- Angular no tiene una definición especifica para los servicios, no hay una clase base o anotación, ni un lugar para registrarlos.
- Los servicios son fundamentales para cualquier aplicación Angular y los componentes son grandes consumidores de servicios.
- La responsabilidad de un componente es implementar la experiencia de los usuarios y nada más: expone propiedades y métodos para el enlace desde la plantilla, y delega todo lo demás en las directivas (manipulación del DOM) y los servicios (lógica de negocio).

Inyección de dependencia

- La Inyección de Dependencias (DI) es un mecanismo que proporciona nuevas instancias de una clase con todas las dependencias que requiere plenamente formadas.
- La mayoría de dependencias son servicios, y Angular usa la DI para proporcionar nuevos componentes con los servicios ya instanciados que necesitan.
- Gracias a TypeScript, Angular sabe de qué servicios depende un componente con tan solo mirar su constructor.
- El Injector es el principal mecanismo detrás de la DI. A nivel interno, un inyector dispone de un contenedor con las instancias de servicios que crea él mismo. Si una instancia no está en el contenedor, el inyector crea una nueva y la añade al contenedor antes de devolver el servicio a Angular, por eso los servicios son singletons en el ámbito de su inyector.
- El provider es cualquier cosa que puede crear o devolver un servicio, típicamente, la propia clase que define el servicio. Los providers pueden registrarse en cualquier nivel del árbol de componentes de la aplicación a través de los metadatos de componentes.

Web Components

- Los Web Components nos ofrecen un estándar que va enfocado a la creación de todo tipo de componentes utilizables en una página web, para realizar interfaces de usuario y elementos que nos permitan presentar información (o sea, son tecnologías que se desarrollan en el lado del cliente). Los propios desarrolladores serán los que puedan, en base a las herramientas que incluye Web Components crear esos nuevos elementos y publicarlos para que otras personas también los puedan usar.
- Los Web Components son una reciente incorporación al HTML5 que, si siguen evolucionando al ritmo al que lo están haciendo, pueden suponer el mayor cambio en el mundo web en años y solucionar de golpe varios problemas históricos de HTML.
- El estándar, en proceso de definición, en realidad, se compone de 4 subelementos complementarios, pero independientes entre si:
  - Custom Elements http://w3c.github.io/webcomponents/spec/custom/
  - Templates https://www.w3.org/TR/html5/scripting-1.html#the-template-element
  - Shadow DOM https://www.w3.org/TR/shadow-dom/
  - HTML Imports http://w3c.github.io/webcomponents/spec/imports/
- Cualquiera de ellos puede ser usado por separado, lo que hace que la tecnología de los Web Elements sea, además de muy útil, muy flexible.

Web Components

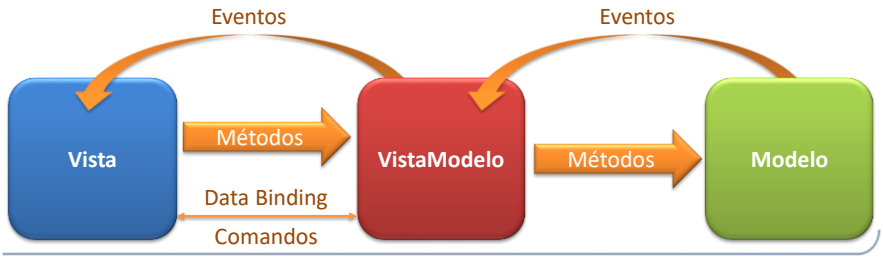
- **Templates**: son fragmentos de HTML que representan al componente de modo que cuando se haga referencia al componente se usará esta plantilla.
- **HTML Import**: es la posibilidad de incluir un HTML dentro de otro mediante un tag <link rel=”import” href=”include.html”>. De este modo el template de un Web Component formará parte del otro.
- **Shadow DOM**: permite encapsular el subarbol DOM de un componente, de modo que la interacción desde fuera y hacia dentro está controlada. Es importante para que los componentes no entren en conflictos al formar parte de la misma página.
- **Custom Elements**: para la definición de Tags específicos que representan a los web components que se han creado.

Angular Elements

- Los Angular Elements son componentes Angular empaquetados como Web Components, también conocidos como elementos personalizados, el estándar web para definir nuevos elementos HTML de una forma independiente del framework.
- Los Web Components son una característica de la Plataforma Web actualmente compatible con las últimas versiones Chrome, Opera y Safari, y están disponibles para otros navegadores a través de polyfills. Un elemento personalizado extiende HTML al permitir definir una etiqueta cuyo contenido está creado y controlado por código JavaScript. El navegador mantiene una serie CustomElementRegistry de elementos personalizados definidos (o componentes web), que asigna una clase de JavaScript instanciable a una etiqueta HTML.
- El paquete @angular/elements exporta una API createCustomElement() que proporciona un puente desde la interfaz de componente y la funcionalidad de detección de cambios de Angular a la API DOM integrada.
- Transformar un componente en un elemento personalizado hace que toda la infraestructura Angular requerida esté disponible para el navegador. La creación de un elemento personalizado es simple y directa, y conecta automáticamente la vista definida por el componente con la detección de cambios y el enlace de datos, asignando la funcionalidad Angular a los equivalentes de HTML nativos correspondientes.

Model View ViewModel (MVVM)

- El **Modelo** es la entidad que representa el concepto de negocio.
- La **Vista** es la representación gráfica del control o un conjunto de controles que muestran el Modelo de datos en pantalla.
- La **VistaModelo** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



Soporte de navegador

Navegador	Versiones compatibles
Chrome	Última
Firefox	Última
Edge	Últimas 2 versiones
IE	11, 10, 9
IE Mobile	11
Safari	Últimas 2 versiones
iOS	Últimas 2 versiones
Android	Nougat (7.0) Marshmallow (6.0) Lollipop (5.0, 5.1) KitKat (4.4)

HERRAMIENTAS DE DESARROLLO

IDEs

- Visual Studio Code - http://code.visualstudio.com/
  - VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.
- StackBlitz - https://stackblitz.com
  - The online IDE for web applications. Powered by VS Code and GitHub.
- Angular IDE by Webclipse - https://www.genuitec.com/products/angular-ide
  - Built first and foremost for Angular. Turnkey setup for beginners; powerful for experts.
- IntelliJ IDEA - https://www.jetbrains.com/idea/
  - Capable and Ergonomic Java * IDE
- Webstorm - https://www.jetbrains.com/webstorm/
  - Lightweight yet powerful IDE, perfectly equipped for complex client-side development and server-side development with Node.js

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente “Ejecutar como Administrador” para evitar otros problemas.
- En algunos casos el firewall de Windows, la configuración del proxy y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar: https://git-scm.com/
- Verificar desde consola de comandos:
  - git

Node.js: Entorno en tiempo de ejecución

- Descargar e instalar: https://nodejs.org
- Verificar desde consola de comandos:
  - node --version

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
  - npm update -g npm
- Verificar desde consola de comandos:
  - npm --version
- Configuración:
  - npm config edit
  - proxy=http://usr:pwd@proxy.dominion.com:8080 ← Símbolos: %HEX ASCII
- Generar fichero de dependencias package.json:
  - npm init
- Instalación de paquetes:
  - npm install -g grunt-cli grunt-init karma karma-cli ← Global (CLI)
  - npm install jasmine-core tslint --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

Generación del esqueleto de aplicación

- Configurar un nuevo proyecto de Angular 2 es un proceso complicado y tedioso, con tareas como:
  - Crear la estructura básica de archivos y bootstrap
  - Configurar SystemJS o WebPack para transpilar el código
  - Crear scripts para ejecutar el servidor de desarrollo, tester, publicación, …
- Disponemos de diferentes opciones de asistencia:
  - Proyectos semilla (seed) disponibles en github
  - Generadores basados en Yeoman
  - Herramienta oficial de gestión de proyectos, Angular CLI.
- Angular CLI, creada por el equipo de Angular, es una *Command Line Interface* que permite generar proyectos y plantillas de código desde consola, así como ejecutar un servidor de desarrollo o lanzar los tests de la aplicación. (https://cli.angular.io/)
  - npm install -g @angular/cli@latest

Proyectos semilla

- Proyectos semilla:
  - https://github.com/angular/quickstart
  - http://mgechev.github.io/angular2-seed
  - https://github.com/ghpabs/angular2-seed-project
  - https://github.com/cureon/angular2-sass-gulp-boilerplate
  - https://angularclass.github.io/angular2-webpack-starter
  - https://github.com/LuxDie/angular2-seed-jade
  - https://github.com/justindujardin/angular2-seed
- Generadores de código basados en Yeoman (http://yeoman.io/generators):
  - https://github.com/FountainJS/generator-fountain-angular2
  - https://github.com/ericmdantas/generator-ng-fullstack

- NOTA: Es conveniente verificar si están actualizados a la última versión de Angular 2, pueden estar desactualizados.

Creación y puesta en marcha

- Acceso al listado de comandos y opciones
  - $ ng help
- Nuevo proyecto
  - $ ng new myApp
  - Esto creará la carpeta myApp con un esqueleto de proyecto ya montado según la última versión de Angular y todo el trabajo sucio de configuración de WebPack para transpilar el código y generar un bundle, configuración de tests, lint y typescript, etc.
  - Además, instala todas las dependencias necesarias de npm e incluso inicializa el proyecto como un repositorio de GIT.
- Servidor de desarrollo
  - ng serve
  - Esto lanza tu app en la URL http://localhost:4200 y actualiza el contenido cada vez que guardas algún cambio.

Generar código

- Hay elementos de código que mantienen una cierta estructura y no necesitas escribirla cada vez que creas un archivo nuevo.
- El comando generate permite generar algunos de los elementos habituales en Angular.
  - Componentes: ng generate component my-new-component
  - Directivas: ng g directive my-new-directive
  - Pipe: ng g pipe my-new-pipe
  - Servicios: ng g service my-new-service
  - Clases: ng g class my-new-class
  - Interface: ng g interface my-new-interface
  - Enumerados: ng g enum my-new-enum
  - Módulos: ng g module my-module
  - Guardian (rutas): ng generate guard my-guard

  --module my-module Indica el modulo si no es el principal
  --project my-lib Indica el proyecto si no es el principal

Schematics en Angular CLI (v.6)

- Schematics es una herramienta de flujo de trabajo para la web moderna; permite aplicar transformaciones al proyecto, como crear un nuevo componente, actualizar el código para corregir cambios importantes en una dependencia, agregar una nueva opción o marco de configuración a un proyecto existente.
- Generadores de componentes de arranque Angular Material:
  - Un componente de inicio que incluye una barra de herramientas con el nombre de la aplicación y la navegación lateral:
    - ng generate @angular/material:material-nav --name=my-nav
  - Un componente de panel de inicio que contenga una lista de tarjetas de cuadrícula dinámica:
    - ng generate @angular/material:material-dashboard --name=my-dashboard
  - Un componente de tabla de datos de inicio que está pre configurado con un datasource para ordenar y paginar:
    - ng generate @angular/material:material-table --name=my-table

Nuevos comandos (v.6)

- **ng update <package>** analiza el package.json de la aplicación actual y utiliza la heurísticas de Angular para recomendar y realizar las actualizaciones que necesita la aplicación.
  - npm install -g @angular/cli
  - npm install @angular/cli
  - ng update @angular/cli
- **ng add <package>** permite agregar nuevas capacidades al proyecto y puede actualizar el proyecto con cambios de configuración, agregar dependencias adicionales o estructurar el código de inicialización específico del paquete.
  - ng add @angular/pwa - Convierte la aplicación en un PWA agregando un manifiesto de aplicación y un service worker.
  - ng add @ng-bootstrap/schematics - Agrega ng-bootstrap a la aplicación.
  - ng add @angular/material - Instala y configura Angular Material y el estilo, y registrar nuevos componentes de inicio en ng generate.
  - ng add @angular/elements - Agrega el polyfill document-register-element.js y las dependencias necesarios para los Angular Elements.

Pruebas

- Son imprescindibles en entornos de calidad: permite ejecutar las pruebas unitarias, las pruebas de extremo a extremo o comprobar la sintaxis.
- Para comprobar la sintaxis: Puedes ejecutar el analizador con el comando:
  - ng lint
- Para ejecutar tests unitarios: Puedes lanzar los tests unitarios con karma con el comando:
  - ng test
- Para ejecutar tests e2e: Puedes lanzar los tests end to end con protractor con el comando:
  - ng e2e

Despliegue

- Construye la aplicación en la carpeta /dist
  - ng build
  - ng build --dev
- Paso a producción, construye optimizándolo todo para producción
  - ng build --prod
  - ng build --prod --env=prod
  - ng build --target=production --environment=prod
- Precompila la aplicación
  - ng build --prod --aot

Estructura de directorios de soporte

- src
- e2e
  - app.e2e-spec.ts
  - app.po.ts
  - tsconfig.json
- .editorconfig
- .gitignore
- angular.json
- karma.conf.js
- package.json
- protractor.conf.js
- README.md
- tsconfig.json
- tslint.json

- src: Fuentes de la aplicación
- e2e: Test end to end
- Ficheros de configuración de librerías y herramientas
- Posteriormente aparecerán los siguientes directorios:
  - node_modules: Librerías y herramientas descargadas
  - dist: Resultado para publicar en el servidor web
  - coverage: Informes de cobertura de código

Estructura de directorios de aplicación

- app
  - app.component.css
  - app.component.html
  - app.component.spec.ts
  - app.component.ts
  - app.module.ts
  - index.ts
- assets
- environments
  - environment.prod.ts
  - environment.ts
- favicon.ico
- index.html
- main.ts
- polyfills.ts
- styles.css
- test.ts
- tsconfig.app.json
- tsconfig.spec.json
- typings.d.ts

- app: Carpeta que contiene los ficheros fuente principales de la aplicación.
- assets: Carpeta con los recursos que se copiaran a la carpeta build.
- environments: configuración de los diferentes entornos.
- main.ts: Arranque del módulo principal de la aplicación. No es necesario modificarle.
- favicon.ico: Icono para el navegador.
- index.html: Página principal (SPA).
- style.css: Se editará para incluir CSS global de la web, se concatena, minimiza y enlaza en index.html automáticamente.
- test.ts: pruebas del arranque.
- polyfills.js: importación de los diferentes módulos de compatibilidad ES6 y ES7.
- Tsconfig.xxxx.json: Configuración del compilador TypeScript para la aplicación y las pruebas.

Webpack

- Webpack (https://webpack.github.io/) es un empaquetador de módulos, es decir, permite generar un archivo único con todos aquellos módulos que necesita la aplicación para funcionar.
- Toma módulos con dependencias y genera archivos estáticos correspondientes a dichos módulos.
- Webpack va mas allá y se ha convertido en una herramienta muy versátil. Entre otras cosas, destaca que:
  - Puede generar solo aquellos fragmentos de JS que realmente necesita cada página.
    - Dividir el árbol de dependencias en trozos cargados bajo demanda
    - Haciendo más rápida la carga inicial
  - Tiene varios loaders para importar y empaquetar también otros recursos (CSS, templates, …) así como otros lenguajes (ES6 con Babel, TypeScript, SaSS, etc).
  - Sus plugins permiten hacer otras tareas importantes como por ejemplo minimizar y ofuscar el código.

Librerías de terceros

- Descargar
  - npm install bootstrap --save
- Referenciar código en angular.json

```
"scripts": [
  "node_modules/jquery/dist/jquery.js",
  {"input": "./node_modules/bootstrap/dist/js/bootstrap.js"},
]
```

- Referenciar estilos en angular.json

```
"styles": [
  {"input": "./node_modules/bootstrap/dist/css/bootstrap.css"},
  "styles.css"
]
```

- Inclusión de ficheros:

```
"assets": [
          "src/assets",
          "src/favicon.ico",
          { "glob": "**/*", "input": "./node_modules/bootstrap/dist/font", "output": "/" }
        ],
```


Glifos

- Font Awesome (https://fontawesome.com/)
- Instalar:

  npm install --save @fortawesome/fontawesome-free
- Configurar en angular.json

```
"build": {
  "options": {
    "styles": [
      "node_modules/@fortawesome/fontawesome-free/css/all.css"
      "styles.css"
    ],
  }
}
```

- Probar:

```
<h1> {{title}} <i class="fa fa-check"></i></h1>
```

GIT

- Preséntate a Git
  - git config --*global* user.name "Your Name Here“
  - git config --*global* user.email your_email@youremail.com
- Crea un repositorio central
  - https://github.com/
- Conecta con el repositorio remoto
  - git remote add origin https://github.com/username/myproject.git
  - git push -u origin master
- Actualiza el repositorio con los cambios:
  - git commit -m "first commit"
  - git push
- Para clonar el repositorio:
  - git clone https://github.com/username/myproject.git *local-dir*
- Para obtener las últimas modificaciones:
  - git pull

MÓDULOS

Módulos

- Los módulos son una buena manera de organizar la aplicación y extenderla con las capacidades de las bibliotecas externas.
- Los módulos Angular consolidan componentes, directivas y pipes en bloques cohesivos de funcionalidad, cada uno centrado en un área de características, un dominio de negocio de la aplicación, un flujo de trabajo o una colección común de los servicios públicos.
- Los módulos pueden añadir servicios a la aplicación, que pueden ser desarrollados internamente, tales como el registrador de aplicación, o pueden provenir de fuentes externas, tales como el enrutador de Angular y el cliente HTTP.
- Muchas bibliotecas de Angular son módulos (por ejemplo: FormsModule, HttpModule, RouterModule).
- Muchas bibliotecas de terceros están disponibles como módulos Angular (por ejemplo: Bootstrap, Material Design, Ionic, AngularFire2).
- Los módulos pueden ser cargados cuando se inicia la aplicación o de forma asíncrona por el router.

Módulo

- Un módulo Angular es una clase adornada con la función decoradora @NgModule: objeto de metadatos que indica cómo Angular compila y ejecuta código del módulo.
  - Declara qué componentes, directivas y pipes pertenecen al módulo.
  - Expone algunos de ellos de manera pública para que puedan ser usados externamente por otras plantillas.
  - Importar otros módulos con los componentes, las directivas y las pipes necesarios para los componentes de este módulo.
  - Se pueden agregar proveedores de servicios a los inyectores de dependencia de aplicación que cualquier componente de la aplicación podrá utilizar.
- Cada aplicación Angular tiene al menos un módulo, el módulo raíz, convencionalmente denominado AppModule. Es el responsable del arranque (Bootstrap) de la aplicación.
- El módulo principal es todo lo que se necesita en una aplicación sencilla con unos pocos componentes.
- A medida que crece la aplicación, se refactoriza el módulo raíz en módulos de características, los cuales representan colecciones de funcionalidad relacionada, que luego se importan al módulo de raíz .

@NgModule

import { NgModule } from '@angular/core';

- **imports**: Especifica una lista de módulos cuyos componentes / directivas / pipes deben estar disponibles para las plantillas de este módulo.
- **declarations**: Especifica una lista de componentes / directivas / pipes que pertenecen a este módulo (son privadas al módulo si no se exportan).
- **providers**: Define el conjunto de objetos inyectables que están disponibles en el inyector de este módulo.
- **exports**: Especifica una lista de componentes / directivas / pipes que se pueden utilizar dentro de las plantilla de cualquier componente que importe este módulo Angular.
- **bootstrap**: Define los componentes que deben ser arrancados cuando se arranque el módulo, se añadirán automáticamente a entryComponents.

Módulo de raíz

```
import { NgModule }     from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

- BrowserModule: módulo necesario para las aplicaciones de navegador.
  - Registra los proveedores de servicios de aplicaciones críticas.
  - También incluye directivas/pipes comunes como NgIf y NgFor que se convierten inmediatamente en visibles y utilizables por cualquiera de las plantillas de los componentes del módulo.

Módulos TypeScript/JavaScript(ES6)

- No confundir con los módulos de Angular.
- Es necesario importar todas las clases, funciones, … que se vayan a utilizar en el código:

```
import { Component, OnInit } from '@angular/core';
import { AppComponent } from './app.component';
```

- Se pueden importar todos los elementos del módulo {*} aunque no suele ser recomendable.
- Hay que referenciar la ruta relativa al fichero del módulo sin extensión.
- Para poder importar algo es necesario haberlo marcado previamente como importable con export.

```
export class AppComponent {
```


Módulos de características

- Generar el módulo:

```
ng generate module myCore
```

- Importar CommonModule en vez de BrowserModule

```
import { CommonModule } from '@angular/common';
```

- Implementar los diferentes elementos del módulo
- Exportar lo elementos compartidos:

```
@NgModule({  // …
  exports: [MyCoreComponent, MyCorePipe]
}) export class MyCoreModule { }
```

- Importar el módulo:

```
import { MyCoreModule }   from './my-core/my-core.module';
@NgModule({ //…
    imports: [ //…
        MyCoreModule]
}) export class AppModule { }
```

- Utilizar las nuevas características obtenidas.

Módulos de características

- Hay cinco categorías generales de módulos de características que tienden a pertenecer a los siguientes grupos:
  - Módulos de características de dominio: ofrecen una experiencia de usuario dedicada a un dominio de aplicación particular, exponen el componente superior y ocultan los subcomponentes.
  - Módulos de características enrutados: los componentes principales son los objetivos de las rutas de navegación del enrutador.
  - Módulos de enrutamiento: proporciona la configuración de enrutamiento para otro módulo y separa las preocupaciones de enrutamiento de su módulo complementario.
  - Módulos de características del servicio: brindan servicios de utilidad como acceso a datos y mensajería.
  - Módulos de características de widgets: hace que los componentes, las directivas y los pipes estén disponibles para los módulos externos (bibliotecas de componentes de UI)

Prevenir la reimportación

- Hay módulos que solo se deben importar en el módulo raíz y una sola vez.
- Para prevenir la reimportación es necesario crear el siguiente constructor en el módulo:

```
constructor (@Optional() @SkipSelf() parentModule: MyCoreModule) {
  if (parentModule) {
    throw new Error(
      'MyCoreModule is already loaded. Import it in the AppModule only');
  }
}
```

- La inyección podría ser circular si Angular busca MyCoreModule en el inyector actual. El decorador @SkipSelf significa “buscar MyCoreModule en un inyector antecesor, por encima de mí en la jerarquía del inyector“.
- Si el inyecto no lo encuentra, como debería ser, suministrará un null al constructor. El decorador @Optional permite que el constructor no reciba el parámetro.

Arranque de la aplicación (v4)

```
// main.ts
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-
browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```


SPA (v4)

```
<!doctype html>
<html lang="en">

<head>
   <meta charset="utf-8">
   <title>Curso de Angular2</title>
   <base href="/">

   <meta name="viewport" content="width=device-width, initial-scale=1">
   <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>

<body>
   <app-root></app-root>
</body>
</html>
```

Módulos usados frecuentemente

NgModule	Importar desde	Por qué lo usas
BrowserModule	@angular/platform-browser	Para ejecutar la aplicación en un navegador
CommonModule	@angular/common	Para usar directivas comunes como NgIf y NgFor
FormsModule	@angular/forms	Cuando crea formularios controlados por plantilla (incluye NgModel)
ReactiveFormsModule	@angular/forms	Al construir formularios reactivos
RouterModule	@angular/router	Para enrutamiento y cuando se quiere utilizar RouterLink, .forRoot() y .forChild()
HttpClientModule	@angular/common/http	Para acceder a un servidor

SERVICIOS

Servicios

- Los servicios en Angular es una categoría muy amplia que abarca cualquier valor, función o característica que necesita una aplicación.
- Angular no tiene una definición especifica para los servicios, no hay una clase base ni un lugar para registrarlos, son simples Clases.
- Tienen la siguiente particularidad:
  - @Injectable: Este decorador avisa a Angular de que el servicio espera utilizar otros servicios y genera los metadatos que necesita el servicio para detectar la Inyección de Dependencias (DI) en el constructor. No es obligatorio ponerlo si el servicio no tiene IoC de otros servicios, pero es recomendable para evitar errores si en el futuro se añade alguna dependencia.
  - Dependency Injection (DI) en el Constructor: Aquí aprovechamos las bondades de TypeScript para pasar explícitamente un objeto con tipo en el constructor. De este modo, Angular es capaz de inferir la Inyección de Dependencias.

Servicio

```
ng generate service datos

import {Injectable} from '@angular/core';
import { Logger } from '../core/logger.service';

@Injectable()
export class DatosService {
   modelo = {};

   constructor(public logger: Logger){}

   metodo1() { … }
   metodo2() { … }
}
```

Inyección de dependencia

- La Inyección de Dependencias (DI) es un mecanismo que proporciona nuevas instancias de una clase con todas las dependencias que requiere plenamente formadas.
- La mayoría de dependencias son servicios, y Angular usa la DI para proporcionar nuevos componentes con los servicios ya instanciados que necesitan.
- Gracias a TypeScript, Angular sabe de qué servicios depende un componente con tan solo mirar su constructor.
- El Injector es el principal mecanismo detrás de la DI. A nivel interno, un inyector dispone de un contenedor con las instancias de servicios que crea él mismo. Si una instancia no está en el contenedor, el inyector crea una nueva y la añade al contenedor antes de devolver el servicio a Angular, por eso se dice que los servicios son singletons en el ámbito de su inyector.
- El provider es cualquier cosa que puede crear o devolver un servicio, como la propia clase que define el servicio. Los providers pueden registrarse en cualquier nivel del árbol de componentes de la aplicación a través de los metadatos de componentes.

Registrar en el inyector

- Globalmente, único para todo el módulo:

```
import { DatosService }  from './datos.service';

@NgModule({
  …
  providers: [DatosService, Logger],
  …
})
export class AppModule { }
```

- En el componente, solo estará disponibles para el componente y su contenido:

```
import { DatosService }  from './datos.service';

@Component({
  …
  providers: [DatosService, Logger],
  …
})
export class AppComponent {
```

Proveedores

- Un proveedor proporciona la versión concreta en tiempo de ejecución de un valor de dependencia. El inyector se basa en los proveedores para crear instancias de los servicios que inyecta en componentes y otros servicios.
- Proveedor simplificado:
  providers: [MyService]
- Proveedor de objeto literal:
  providers:[{ provide: alias, **useClass**: MyService}]
- Una instancia para dos proveedores:
  providers: [NewLogger, { provide: OldLogger, **useExisting**: NewLogger}]
- Añadir a un proveedor múltiple:
  { provide: NG_VALIDATORS, **useExisting**: MyDirective**, multi: true** }
- Proveedor de valor ya instanciado:
  { provide: VERSION, **useValue**: '2.1.0.234' }]

Factorías

- A veces tenemos que crear el valor dependiente dinámicamente, en base a información que no tendremos hasta el último momento posible. Tal vez la información cambia varias veces en el transcurso de la sesión de navegación. Supongamos también que el servicio inyectable no tiene acceso independiente a la fuente de esta información.
- Esta situación requiere de un proveedor factoría: función que genera la instancia del servicio.

```
export class DatosService {
  constructor(logger: Logger, isAuthorized: boolean){}
  // …
}
export let DatosServiceFactory = (logger: Logger, userService: UserService) => {
  return new DatosService(logger, userService.user.isAuthorized);
};
providers: [{provide: DatosService, useFactory: DatosServiceFactory,
    deps: [Logger, UserService] }]
```

Auto referenciado de proveedores (v.6)

- Se puede especificar que el servicio debe proporcionarse en el inyector raíz: agrega un proveedor del servicio al inyector del módulo principal, estos proveedores están disponibles para todas las clases en la aplicación, siempre que tengan el token de búsqueda.

```
@Injectable({
  providedIn: 'root',
})
```

- También es posible especificar que se debe proporcionar un módulo en particular:

```
@Injectable({
  providedIn: MyCoreModule,
})
```

- Dan a Angular la capacidad de eliminar servicios del producto final que no se usan en su aplicación.

Dependencias que no son de clase

- La inyección utiliza los nombres de las clase. A veces se quiere inyectar algo que no tiene una representación en tiempo de ejecución: interfaces, genéricos, arrays, …
- En dichos casos es necesario crear una cadena o un token y referenciar con el decorador @Inject.
- Mediante cadenas:

```
providers: [{ provide: 'VERSION', useValue: '2.1.0.234' }]
constructor(@Inject('VERSION') public version: string)
```

- InjectionToken<T> permite crear un token que se puede utilizar en un proveedor de DI, donde T que es el tipo de objeto que será devuelto por el Inyector. Esto proporciona un nivel adicional de seguridad de tipo.

```
export const VERSION = new InjectionToken<string>('Version');
providers: [{ provide: VERSION, useValue: '2.1.0.234' }]
constructor(@Inject(VERSION) public version: string)
```

Dependencias opcionales

- Se puede indicar que se inyecte solo si existe la clase del servicio:

```
import { Optional } from '@angular/core';

class AnotherService{
   constructor(@Optional() private logger: Logger) {
     if (this.logger) {
       this.logger.log("I can log!!");
     }
   }
}
```

- Es importante que el componente o servicio este preparado para que su dependencia tenga valor null.

Inyectores Explícitos

- Habitualmente no tenemos que crear los inyectores en Angular, dado que se crea automáticamente un inyector a nivel del módulo para toda la aplicación durante el proceso de arranque.
- Los componentes cuentan con sus propios inyectores.
- Se puede crear un inyector de forma explicita:
  - injector = Injector.create([Http, Logger]);
- Para posteriormente obtener instancias inyectadas:
  - let srv= injector.get(Logger);
- Las dependencias son únicas dentro del alcance de un inyector, sin embargo, al ser Angular DI un sistema de inyección jerárquica, los inyectores anidados pueden crear sus propias instancias de servicio.

Jerárquica de inyectores de dependencia

- Angular tiene un sistema de inyección de dependencia jerárquica. Hay un árbol de inyectores que se asemeja al árbol de componentes de una aplicación. Se pueden configurar los inyectores en cualquier nivel de ese árbol de componentes.
- De hecho, no existe el inyector. Una aplicación puede tener múltiples inyectores. Una aplicación angular es un árbol de componentes, cada instancia de componente tiene su propio inyector, que es paralelo al árbol de los inyectores.
- Cuando un componente solicita una dependencia, Angular intenta satisfácela con un proveedor registrado en el propio inyector de ese componente. Si el componente carece del proveedor, transfiere la solicitud al inyector de su componente anfitrión. Las solicitudes siguen burbujeando hasta que Angular encuentra un inyector que puede manejar la solicitud o llega al módulo. Si se queda sin antepasados y no está registrado en el módulo, Angular genera un error.
- La anotación @Host() impide el burbujeo y obliga a que el componente anfitrión tenga registrado el proveedor.

```
constructor(@Host() public logger: Logger){}
```


Servicios importados

- Un módulo importado que añade proveedores a la aplicación puede ofrecer facilidades para la configuración de dichos proveedores.
- Por convención, el método estático forRoot de la clase módulo proporciona y configura al mismo tiempo los servicios. Recibe un objeto de configuración de servicio y devuelve un objeto ModuleWithProviders:

```
static forRoot(config: MyServiceConfig): ModuleWithProviders {
  return {
    ngModule: CoreModule,
    providers: [{provide: MyServiceConfig, useValue: config }]
  };
}
```

- En el servicio:

```
constructor(@Optional() config: MyServiceConfig) {
  if (config) { … }
}
```

- En el módulo principal:

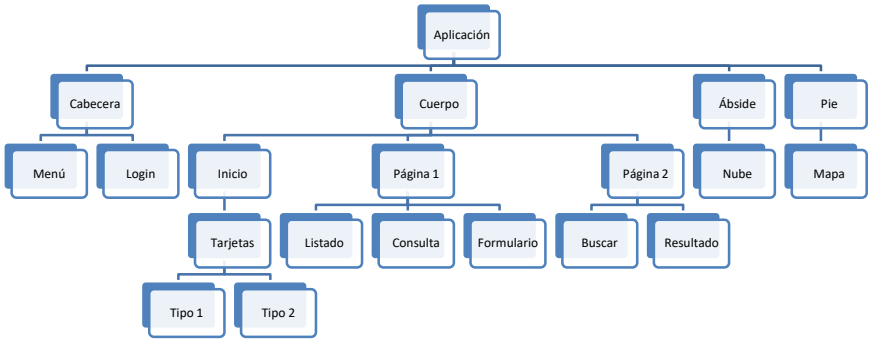
```
imports: [ //…
   MyCoreModule.forRoot({…}),
],
```

COMPONENTES

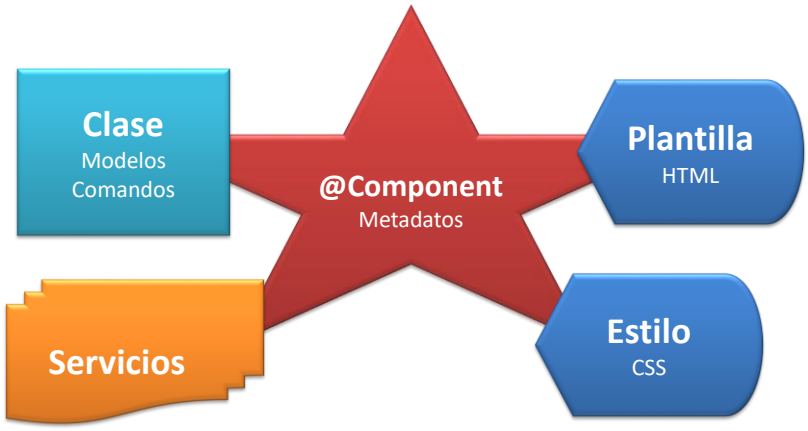
Modelo de componentes

- Los componentes encapsulan el contenido (clase), la funcionalidad (clase), la presentación (plantilla) y la estética (css) de un elemento visual.
- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente.
- Sigue un modelo de composición jerárquico con forma de árbol de componentes.
- La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Los componentes establecen un cauce bien definido de entrada/salida para su comunicación con otros componentes.
- Los componentes son clases:
  - Decoradas con @Component
  - Que exponen datos (modelos) y funcionalidad (comandos) para su consumo por la plantilla
  - Opcionalmente, exponen propiedades (@Input) y eventos (@Output) para interactuar con otros componentes.

Árbol de componentes



Componente



@Component

- **selector**: Selector CSS (Tag HTML) que indica a Angular que debe crear e instanciar el componente cuando se encuentra un elemento con ese nombre en el HTML.
- **template**: Cadena con el contenido de la plantilla o
- **templateUrl**: La url en la que se encuentra plantilla que se quiere vincular al componente.
- **styles**: Cadena con el contenido del CSS o
- **styleUrls**: Lista de urls a archivos de estilos que se aplican al componente (acepta Less, Sass, Stylus).
- **entryComponents**: Lista de los Componentes/Directivas/Pipes que se insertan dinámicamente en la plantilla del el componente.
- **providers**: Lista de los proveedores disponibles para este componente y sus hijos.

Componente

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = 'Hola Mundo';

  constructor() { }

  despide(): void {
    this.title = 'Adios Mundo';
  }
}
```

Registrar

- Globalmente:

```
import { AppComponent }  from './app.component';

@NgModule({
  …
  declarations: [ AppComponent ],
  …
})
export class AppModule { }
```

- En el contenedor:

```
import { HijoComponent }  from './hijo.component';

@Component({
  …
  entryComponents:  [HijoComponent, …],
  …
})
export class AppComponent { … }
```


Propiedades enlazables

- Externamente se comportan como un atributo público.
- Internamente están compuesta por dos métodos, get/set que controlan como entra y sale la información. Las propiedades de solo lectura solo disponen del método get.
- No es obligatorio que tengan un reflejo directo en los atributos de la clase.

```
class EmployeeVM {
   private _fullName: string;
   get fullName(): string { return this._fullName;  }
   set fullName(newName: string) {
      if (this.validate(newName)) {
         this._fullName = newName;
         this.fullNameChanged();
      } else {
        // …
      }
   }
   // …
}
let employee = new EmployeeVM(); employee.fullName = "Bob Smith";
```

Atributos del selector

- Los atributos del selector del componente se comportan como los atributos de las etiquetas HTML, permitiendo personalizar y enlazar al componente desde las plantillas.
- Propiedades de entrada

  @Input() init: string;

  <my-comp [init]="1234"></my-comp>
- Eventos de salida

  @Output() updated: EventEmitter<any> = new EventEmitter();

  this.updated.emit(value);

  <my-comp (updated)="onUpdated($event)"></my-comp>
- Propiedades bidireccionales: Es la combinación de una propiedad de entrada y un evento de salida con el mismo nombre (el evento obligatoriamente con el sufijo Change):

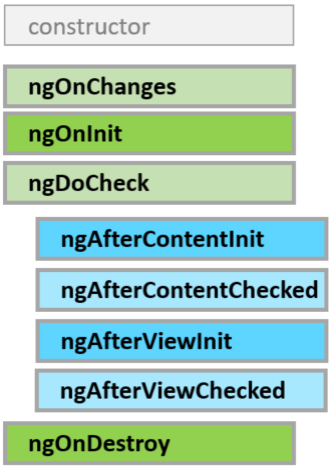
  @Input() size: number | string;

  @Output() sizeChange = new EventEmitter<number>();

Atributos del selector

```
import { Component, EventEmitter, Input, Output } from '@angular/core';
@Component({
 selector: 'my-sizer',
 template: ` <div>
   <button (click)="dec()">-</button><button (click)="inc()">+</button>
   <label [style.font-size.px]="size">FontSize: {{size}}px</label>
 </div>`
})
export class SizerComponent {
 @Input() size: number | string;
 @Output() sizeChange = new EventEmitter<number>();
 dec() { this.resize(-1); }
 inc() { this.resize(+1); }
 resize(delta: number) {
   this.size = Math.min(40, Math.max(8, +this.size + delta));
   this.sizeChange.emit(this.size);
 }
}
<my-sizer [(size)]="fontSizePx"></my-sizer>
<my-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></my-sizer>
```

Ciclo de vida



- Cada componente tiene un ciclo de vida gestionado por el Angular.
- Angular lo crea y pinta, crea y pinta sus hijos, comprueba cuando sus propiedades enlazadas a datos cambian, y lo destruye antes de quitarlo del DOM.
- Angular ofrece ganchos al ciclo de vida que proporcionan visibilidad a dichos momentos clave y la capacidad de actuar cuando se producen.

Ciclo de vida

Gancho	Propósito y temporización
ngOnChanges	Responder cuando Angular (re) establece las propiedades de entrada enlazadas a datos.
ngOnInit	Inicializar el componente después de que Angular muestre las primeras propiedades enlazadas a datos y establece las propiedades de entrada del componente.
ngDoCheck	Llamado cada vez que las propiedades de entrada de un componente o una directiva se comprueban. Lo utilizan para extender la detección de cambios mediante la realización de una comprobación personalizada.
ngAfterContentInit	Responder después de que Angular proyecta el contenido externo en la vista del componente.
ngAfterContentChecked	Responder después de que Angular chequee el contenido proyectado en el componente.
ngAfterViewInit	Responder después de que Angular inicialice las vistas del componente y sus hijos.
ngAfterViewChecked	Responder después de que Angular chequee las vistas del componente y sus hijos.
ngOnDestroy	Limpiar justo antes de que Angular destruya el componente.

ESTILOS

Introducción

- Las aplicaciones Angular utilizan CSS estándar para establecer la estética. Angular puede asociar el estilo al componente, lo que permite un diseño más modular que las hojas de estilo regulares.
- Para cada componente Angular se puede definir no solo una plantilla HTML, sino también los estilos CSS que acompañan a esa plantilla, especificando los selectores, las reglas y las consultas de medios que se necesite.
- Los estilos especificados en los metadatos se aplican solo dentro de la plantilla de ese componente, no son heredados por ningún componente anidado dentro de la plantilla ni por ningún contenido proyectado en el componente.

Modularidad de estilo

- Se pueden usar los nombres y selectores de clases de CSS que tengan más sentido en el contexto de cada componente.
- Los nombres de clase y los selectores son locales para el componente y no colisionan con las clases y los selectores utilizados en otras partes de la aplicación.
- Los cambios en los estilos de otras partes de la aplicación no afectan los estilos del componente.

- Se puede ubicar conjuntamente el código CSS de cada componente con el código de TypeScript y HTML del componente, lo que conduce a una estructura de proyecto prolija y ordenada.
- Se puede cambiar o eliminar el código CSS del componente sin buscar en toda la aplicación para encontrar dónde se usa el código.

Selectores especiales

- :host selector de pseudoclases para el elemento que aloja al componente. No se puede llegar al elemento host desde el interior del componente con otros selectores porque no forma parte de la propia plantilla del componente dado que está en la plantilla de un componente anfitrión.

```
:host { border: 1px solid black; }
:host(.active) { border-width: 3px; }
```

- :host-context() selector que busca una clase CSS en cualquier antecesor del elemento host del componente, hasta la raíz del documento, solo es útil cuando se combina con otro selector para aplicar estilos basados en alguna condición externa al componente (si está contenido dentro de un elemento con determinada clase).

```
:host-context(.theme-light) h2 {
  background-color: #eef;
}
```

Agregar estilos a un componente

- Estilos en los metadatos de los componentes
  styles: ['h1 { font-weight: normal; }']
- Archivos de estilo en los metadatos de los componentes
  styleUrls: ['./my.component.css']
- Etiqueta <style> en la plantilla
  template: `
    <style>…</style>
  `
- Etiqueta <link> en la plantilla (el enlace debe ser relativo a la raíz de la aplicación)
  template: `
    <link rel="stylesheet" href="../assets/my.component.css">
  `
- También puede importar archivos CSS en los archivos CSS usando la regla @import del CSS estándar (la URL es relativa al archivo CSS en el que está importando).

Encapsulación

- Los estilos CSS de los componentes se encapsulan en la vista del componente y no afectan el resto de la aplicación.
- Para controlar cómo ocurre esta encapsulación por componente se puede establecer el modo de encapsulación en los metadatos del componente:
```
@Component({
  styleUrls: ['./app.component.less'],
  encapsulation: ViewEncapsulation.Native
})
```
- **Native**: la encapsulación de vista utiliza la implementación DOM nativa del sombreado del navegador (Shadow DOM) para adjuntar un DOM sombreado al elemento host del componente, y luego coloca la vista de componente dentro de esa sombra DOM. Los estilos del componente se incluyen dentro del DOM sombreado. (Requiere soporte nativo de los navegadores.)
- **Emulated** (por defecto): la encapsulación de vista emula el comportamiento del DOM sombreado mediante el preprocesamiento (y cambio de nombre) del código CSS para aplicar efectivamente el CSS a la vista del componente.
- **None**: no se encapsula la vista. Angular agrega el CSS a los estilos globales. Las reglas, aislamientos y protecciones no se aplican. Esto es esencialmente lo mismo que pegar los estilos del componente en el HTML.

Integración con preprocesadores CSS

- Si se está utilizando AngularCLI, se pueden escribir archivos de estilo en Sass, Less o Stylus y especificar los archivos en @Component.styleUrls con las extensiones adecuadas (.scss, .less, .styl):

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.less']
})
...
```

- El proceso de compilación del CLI ejecutará el preprocesador de CSS pertinente. Esto no es posible con los estilos en línea.

Configuración

- Se puede establecer como se generan por defecto los componentes:

```
"schematics": {
    "@schematics/angular:component": {
      "inlineTemplate": true,
      "viewEncapsulation": "Emulated |Native | None"
      "styleext": "css | less | scss | styl "
    }
  },
...
 "options": {
        "styles": [
          "src/styles.less"
        ],
```

PLANTILLAS

Plantillas

- En Angular, las plantillas se escriben en HTML añadiéndole elementos y atributos específicos Angular.
- Angular combina la plantilla con la información y funcionalidad de la clase del componente para crear la vista dinámica que el usuario visualiza en el navegador.
- Casi toda las etiquetas del body HTML son válidas en las plantillas. La excepción principal es <script> que está prohibida para evitar el riesgo de ataques de inyección de secuencia de comandos.
- El vocabulario HTML se amplia en las plantillas con componentes y directivas que aparecen con la notación de las etiquetas y los atributos.
- La interpolación permite introducir marcadores y pipes.

Interpolación

- Los marcadores Angular son expresiones similares a las de JavaScript que se colocan entre llaves dobles. Aceptan valores vinculados e invocación de funciones.
  {{expresión}}
- Una expresión de plantilla produce un valor. Angular ejecuta la expresión y asigna el resultado a una propiedad de un objetivo vinculado; el objetivo podría ser un elemento HTML, un componente o una directiva.
- Las expresiones Angular son similares a las de JavaScript con las siguientes diferencias:
  - El contexto en Angular es la instancia del componente y la propia plantilla.
  - Están prohibidas las expresiones que tienen o promueven efectos secundarios, incluyendo: asignaciones (=, +=, -=, ...), new, operadores de incremento y decremento (++ y --), y el encadenamiento expresiones con ; o ,
  - No hay soporte para los operadores binarios | y &
  - | y ?. son operadores de expresión con significado propio en Angular.
- Estas limitaciones se pueden salvar creando un método (función) en el componente que obtenga el resultado para ser llamado desde la expresión.
- Se puede utilizar Pipes en las expresiones para dar formato a los datos o transformarlos antes de mostrar el resultado.

Directrices de las expresiones

- **Sin efectos secundarios visibles**: Una expresión de plantilla no debe cambiar ningún estado de aplicación que no sea el valor de la propiedad de destino.
- **Ejecución rápida**: Angular ejecuta expresiones de plantilla después de cada ciclo de detección de cambios. Las expresiones deben finalizar rápidamente o la experiencia del usuario se puede deteriorar, especialmente en los dispositivos más lentos. Se deben cachear los cálculos mas costosos.
- **Simplicidad**: Aunque es posible escribir expresiones bastante complejas, se deben evitar. Por norma deberían ser el nombre de una propiedad o una llamada a un método, para no invadir las responsabilidades de la capa de negocio.
- **Idempotencia**: Una expresión idempotente siempre devuelve exactamente el mismo resultado hasta que cambia uno de sus valores dependientes: están libres de efectos secundarios y mejoran el rendimiento de detección de cambios de Angular.

Enlace de datos

- El enlace de datos automatiza, de una forma declarativa, la comunicación entre las etiquetas HTML de la plantilla y las propiedades y métodos del código del componente.
- La dirección y tipo del enlazado se puede expresar mediante símbolos envolventes (recomendado) o mediante prefijos.

Dirección de los datos	Sintaxis	Tipo de enlace
Unidireccional desde la fuente de datos hacia la plantilla	{{expression}} [target] = "expression" **bind-**target = "expression"	Interpolación Propiedad Atributo Class Style
Unidireccional de la plantilla a la fuente de datos	(target) = "statement" **on-**target = "statement"	Evento
En ambos sentidos	[(target)] = "expression" **bindon-**target = "expression"	Bidireccional

Atributo HTML vs Propiedad DOM

- Los atributos están definidos por el HTML y las propiedades se definen mediante el DOM (Document Object Model):
  - Hay atributos que tienen una correspondencia 1: 1 con propiedades (Ej.: id).
  - Algunos atributos no tienen una propiedad correspondiente (Ej.: colspan).
  - Algunas propiedades no tienen un atributo correspondiente (Ej.: textContent).
  - Muchos atributos HTML parecen correlacionarse con propiedades ... pero no en la forma en que podríamos pensar!
- El atributo HTML y la propiedad DOM nunca son lo mismo, incluso cuando tienen el mismo nombre.
- Los valores de una propiedad pueden cambiar, mientras que el valor del atributo no puede cambiar una vez expresado.
- Los atributos pueden inicializar propiedades, siendo las propiedades las que luego cambian.
- La distinción entre ambos es fundamental para entender cómo funcionan los enlaces en Angular: La vinculación se realiza sobre propiedades y eventos del DOM, no sobre atributos (solo son interpolables).

Objetivo de un enlace de datos

Tipo	Objetivo	Ejemplos
Propiedad	Elemento	<img [src] = "heroImageUrl">
	Componentes	<hero-detail [hero]="currentHero"></hero-detail>
	Directiva	<div [ngClass] = "{selected: isSelected}">
Evento	Elemento	<button (click) = "onSave()">Save</button>
	Componentes	<hero-detail (delete)="deleteHero()"></hero-detail>
	Directiva	<div (myClick)="clicked=$event">click me
Bidireccional		<input [(ngModel)]="heroName">
Atributo	Interpolación	<img src = "{{heroImageUrl}}">
Atributo	attr	<button [attr.aria-label]="help">help</button>
Estilo	class	<div [class.special]="isSpecial">Special
Estilo	style	<button [style.color] = "isSpecial ? 'red' : 'green'">

Vinculación

- Para permitir que un elemento de la página se haga visible o invisible en función de un valor del modelo.

```
<input name="nombre" type="text" [hidden]="tieneNombre ===false"/>
```

- Para habilitar o deshabilitar un elemento de entrada de datos como un <input>, un <select> o un <button> en función de una condición.

```
<input type="button" value="Send" [disabled]="!isValid"/>
```

- Para vincular el contenido de la etiqueta en modo textual.

```
<div>{{mensaje}}</div>
<div [textContent]="mensaje"></div>
```

- Para vincular el contenido de la etiqueta en modo HTML.

```
<div [innerHTML]="mensaje"></div>
```

Eventos

- Invocación de un comando:
  <button (click)="onClickMe()">Click me!</button>
- Invocación de un comando con parámetros:
  <button (click)="onClickMe(1,2)">Click me!</button>
- Invocación de una expresión (no recomendable):
  <button (click)="numero=2; otra=dato;">Click me!</button>
- El objeto $event encapsula la información relativa al evento:
  - Si se ha disparado con el método emit del EventEmitter, tendrá el tipo definido en el genérico.
  - Si es un evento del DOM, es un objeto de evento estándar del DOM.
- Invocación de un controlador de eventos:

```
<input (keyup)="onKey($event)">
 onKey(event:any) {
   this.values += event.target.value + ' | ';
 }
```


Directivas

- Las directivas son marcas en los elementos de las plantilla que indican al Angular que debe asignar cierto comportamiento a dichos elementos o transformarlos según corresponda.
- Permiten añadir comportamiento dinámico al árbol DOM.
- La recomendación es que debe ser en las directivas el único sitio donde se manipular el árbol DOM, para que entre dentro del ciclo de vida de compilación, binding y renderización del HTML.
- Los componentes y el nuevo sistema de enlazado han reducido enormemente la necesidad de directivas de versiones anteriores.
- Se clasifican en dos tipos:
  - directivas atributos que alteran la apariencia o el comportamiento de un elemento existente.
  - directivas estructurales que alteran el diseño mediante la adición, eliminación y sustitución de elementos (nodos) del DOM.

Estilos: ngStyle

- Mediante enlazado:

  <div [style.background-color]="canSave ? 'cyan': 'grey'">

  <div [style.font-size.em]="myFontSize">

- ngStyle permite gestionar el estilo dinámicamente:

```
setStyles() {
  let styles = {
    // CSS property names
    'font-style':  this.canSave      ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
    'font-size':   this.isSpecial    ? '24px'   : '8px'
  };
  return styles;
}
<div [ngStyle]="setStyles()">
```


Estilos: ngClass

- Mediante enlazado:

  <div [class]="strClasesCss">

  <div [class.remarcado]="isSpecial">

- ngClass simplifica el proceso cuando se manejan múltiples clases que aparecen y desaparecen:
  - CSS

```
.error { color:red; }
.urgente { text-decoration: line-through; }
.importante { font-weight: bold; }
```

  - TS

```
this.clasesCss={error:!this.isValid,
urgente:this.isSpecial, importante:true }
```

  - HTML

```
<div [ngClass]="clasesCss">texto</div>
```

Enlace bidireccional: ngModel

- Permite el enlace de datos bidireccional con los controles de formulario: input, textarea y select.
- Homogeniza los diferentes mecanismos para representar el valor en los diferentes controles: value, textcontent, checked, selected, …
- La expresión de vinculación debe ser un atributo publico o una propiedad de la clase.

```
<input [(ngModel)]="vm.nombre">
```

- Antes de utilizar la directiva ngModel se debe importar FormsModule en el módulo principal:

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [ BrowserModule,  FormsModule, // …
```


ngModelOptions

```
options: {
   name?: string;
   standalone?: boolean;
   updateOn?: FormHooks;
}
```

- name: una alternativa a establecer el atributo de nombre en el elemento de control de formulario mediante una cadena.
- standalone: a true no se registrará con su formulario principal y actuará como si no estuviera en el formulario para añadir metacontroles al formulario.

```
<input type="checkbox" [ngModel]="opciones"
[ngModelOptions]="{standalone: true}"> ¿Ver mas opciones?
```

- updateOn : Define el evento en el que se validará el control y actualizará el valor del modelo. Por defecto 'change' pero también acepta 'blur' y 'submit'.

Directivas condicionales

- ***ngIf**: determina que exista o no una etiqueta dependiendo de una condición.

```
<div *ngIf="condición">
```

- **ngSwitch**: determina cual es la etiqueta que va a existir dependiendo de un valor. Esta directiva se complementa con las siguientes:
  - ***ngSwitchCase**: Su valor es un literal o propiedad que se compara con el de la expresión del ngSwitch, si coinciden la etiqueta existirá.
  - ***ngSwitchDefault**: Si no se cumple ninguna condición de los *ngSwitchCase se muestra lo que tenga la directiva *ngSwitchDefault.

```
<div [ngSwitch]="expresion">
    <div *ngSwitchCase="'A'">En valor hay una A</div>
    <div *ngSwitchCase="'B'">En valor hay una B</div>
    <div *ngSwitchDefault>Ni una A ni una B</div>
</div>
```


Sintaxis if…else (v.4)

```
<!-- if/else syntax example -->
<ng-template #forbidden>
  <p>Sorry, you are not allowed to read this content</p>
</ng-template>
<p *ngIf="isAuth; else forbidden;">
  <p>Some secret content</p>
</p>

<!-- Other if/else syntax example -->
<div *ngIf="isAuth; then authenticated; else forbidden;"></div>
<ng-template #authenticated>
  <p>Some secret content</p>
</ng-template>
```

Directiva iterativa: *ngForOf

- Recorre una colección generando la etiqueta para cada uno de sus elementos.
- La cadena asignada a no es una expresión de plantilla. Es una microsintaxis, un pequeño lenguaje propio que Angular interpreta. Utiliza la sintaxis del foreach: "let item of coleccion".
- La palabra clave let permite crea una variable de entrada de plantilla.
- Otros posibles valores para las variables de entrada de plantilla son:
  - index: Un número que indica el nº de elementos(de 0 a n-1).
  - first / last: Vale true si es el primer o el último elemento del bucle.
  - even / odd : Vale true si es un elemento con $index par o impar (0 es par).

```
<tr *ngFor="let p of provincias; let ind=index; let fondo=odd;"
    [style.background-color]="fondo ? 'LightBlue' : 'Lavender'">
     <td>{{ind}}</td>
     <td>{{p.id}}</td>
     <td>{{p.nombre}}</td>
</tr>
```


Directiva iterativa: *ngForOf

- ngFor por defecto controla los elementos de la lista usando la identidad del objeto.
- Si cambian las identidades de los datos, Angular automáticamente borrará todos los nodos DOM y los volverá a crear con los nuevos datos.
- Esto puede ser poco eficiente, un pequeño cambio en un elemento, un elemento eliminado o agregado puede desencadenar una cascada de manipulaciones de DOM.
- El seguimiento por identidad de objeto es una buena estrategia predeterminada porque Angular no tiene información sobre el objeto, por lo que no puede decir qué propiedad debe usar para el seguimiento.
- Se puede solucionar con una función que recibe el índice y el elemento, devolviendo la nueva identidad.

```
*ngFor="let p of provincias; let ind=index; trackBy: trackByFn"
trackByFn(index, item) { return item ? item.id : undefined; }
```

Directiva iterativa: *ngForOf

- Para rellenar un select (anteriormente ng-options):

```
<select [(ngModel)]="idProvincia" >
  <option value="">--Elige opción--</option>
  <option *ngFor="let p of provincias"
          [value]="p.id">{{p.nombre}}</option>
</select>
```

- Las directivas *ngFor, *ngIf, y ngSwitch, a nivel interno, se envuelven en una directiva <ng-template>.
- Se pueden implementar directamente sin el *:

```
<p>
    <ng-template ngFor let-provincia [ngForOf]="provincias"
          [ngForTrackBy]="trackByProvincias">
     {{provincia.nombre}}<br>
    </ng-template>
</p>
```


ng-container (v.4)

- La directiva <ng-container> es un elemento de agrupación que proporciona un punto donde aplicar las directivas sin interferir con los estilos o el diseño porque Angular no la refleja en el DOM.
- Una parte del párrafo es condicional:

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

- No se desea mostrar todos los elemento como opciones:

```
<select [(ngModel)]="hero">
  <ng-container *ngFor="let h of heroes">
    <ng-container *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </ng-container>
  </ng-container>
</select>
```

*ngComponentOutlet

- Permite instanciar e insertar un componente en la vista actual, proporcionando un enfoque declarativo a la creación de componentes dinámicos.

  <ng-container *ngComponentOutlet="componentTypeExpression*;*
  injector: injectorExpression; content: contentNodesExpression; ngModuleFactory: moduleFactory;"></ng-container>

  - Se puede controlar el proceso de creación de componentes mediante los atributos opcionales:
    - injector: inyector personalizado que sustituye al del contenedor componente actual.
    - content: Contenido para transcluir dentro de componente dinámico.
    - moduleFactory: permite la carga dinámica de otro módulo y, a continuación, cargar un componente de ese módulo.
- El componentTypeExpression es la clase del componente que debe ser expuesta desde el contexto:

  <ng-container *ngComponentOutlet="componente"></ng-container>
  @Component({entryComponents: [MyComponent //…
  public componente: any = MyComponent;

Variables referencia de plantilla

- Una variable referencia de plantilla es una referencia a un elemento DOM o directiva dentro de una plantilla, similar al atributo ID de HTML pero sin las implicaciones funcionales del mismo.
- Pueden identificar tanto a elementos DOM nativos como a componentes Angular.
- Pueden ser utilizadas a lo largo de toda la plantilla por lo que no deben estar duplicadas.
- Los nombre se definen con el prefijo # o ref-, y se utilizan sin prefijo:

  <input **#phone** placeholder="phone number">
  <button (click)="callPhone(**phone**.value)">Call</button>
  <input **ref-fax** placeholder="fax number">
  <button (click)="callFax(**fax**.value)">Fax</button>
- Pueden tomar el valor de determinadas directivas:

  <form **#theForm="ngForm"** (ngSubmit)="onSubmit(**theForm**)">
  <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
  <input [(ngModel)]="currentHero.firstName" **#firstName="ngModel"**>

Operador de navegación segura ?.

- El operador de navegación segura ?. es una forma fluida y conveniente para protegerse de los valores nulos o no definidos en las rutas de invocación de las propiedades.
- Si se invoca una propiedad de un objeto cuya referencia es nula JavaScript lanza un error de referencia nula y lo mismo ocurre Angular.
- Si title es null o undefined la siguiente interpolación

  The title is {{title}}

  sustituye el marcador por una cadena vacía sin dar error.
- Si obj es null o undefined la siguiente interpolación

  The title is {{obj.title}}

  lanza un error de referencia nula.
- Para evitar el error sustituyendo el marcador por una cadena vacía:

  The title is {{obj?.title}}

Operador de aserción no nulo !.

- A partir de Typescript 2.0, con --strictNullChecks puede aplicar la verificación estricta de nulos, que asegura que ninguna variable sea involuntariamente nula o indefinida al dejarla sin asignar o si se intenta asignar valores nulos o indefinidos si el tipo no lo permite valores nulos ni indefinidos. También da un error si no puede determinar si una variable será nula o indefinida en el tiempo de ejecución.
- Cuando el compilador de Angular convierte la plantilla en código TypeScript, el operador de aserción no nulo le dice al verificador de tipo de TypeScript que suspenda la verificación estricta de nulos para el marcador.
- Si title es null o undefined la siguiente interpolación

  The title is {{obj!.title}}

  sustituye el marcador por una cadena vacía sin dar error.
- A veces, un marcador generará un error de tipo y no es posible o difícil especificar completamente el tipo. Para silenciar el error, se puede utilizar la función $any de conversión:

  The title is {{$any(obj).title}}

Operador de canalización (|)

- El resultado de una expresión puede requerir alguna transformación antes de estar listo para mostrarla en pantalla: mostrar un número como moneda, que el texto pase a mayúsculas o filtrar una lista y ordenarla.
- Los Pipes de Angular, anteriormente denominados filtros, son simples funciones que aceptan un valor de entrada y devuelven un valor transformado. Son fáciles de aplicar dentro de expresiones de plantilla, usando el operador tubería (|).
- Son una buena opción para las pequeñas transformaciones, dado que al ser encadenables un conjunto de transformaciones pequeñas pueden permitir una transformación compleja.

  Birthdate: {{birthdate | date:'longDate' | uppercase}}
- Permiten el paso de parámetros, precedidos de :, para la particularización de la transformación.

Pipes Predefinidos

- **number**: se aplica sobre números para limitar el nº máximo y mínimo de dígitos que se muestran de dicho número, también cambia al formato al idioma actual si lo soporta el navegador.

  number[: {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}]

  ```
  <div{{pi | number:'3.2-5'}}</div>
  ```
- **currency**: muestra un número con el símbolo de la moneda local y con el número de decimales correctos.

  ```
  <div>{{ importe | currency }} {{ importe |
      currency:'EUR':'symbol':'4.2-2' }}</div>
  ```
- **percent**: muestra un número como un porcentaje.

  ```
  <div>{{ ratio | percent}} {{ ratio | percent:'2.2-2'}}</div>
  ```
- **date**: se aplica sobre fechas para mostrar un formato concreto. Se puede expresar mediante marcadores (dd,MM,yyyy,HH,mm, ...) o formatos predefinidos (medium, short, fullDate, longDate, mediumDate, shortDate, mediumTime, shortTime):

  ```
  <div>{{miFecha | date:'dd/MM/yyyy'}}</div>
  ```

Pipes Predefinidos

- **lowercase**: transformará un String a minúsculas.
```
<div>{{ nombre | lowercase }}</div>
```
- **uppercase**: transformará un String a mayúsculas.
```
<div>{{ nombre | uppercase }}</div>
```
- **titlecase:** transformará un String a mayúsculas las iniciales.
```
<div>{{ nombre | titlecase }}</div>
```

- **slice**: extrae una subcadena, si se aplica a String, o un subconjunto de elementos, si se aplica a una lista. El Índice inicial es obligatorio, el final es opcional. Los valores negativos toman como base la última posición.
```
<div>Inicial: {{ nombre | slice:0:1}}</div>
<div>Termina: {{ nombre | slice:-1}}</div>
<ul><li *ngFor="let n of list|slice:1:10">{{n}}</li></ul>
```

- **json**: convierte un objeto JavaScript en una cadena JSON, solo utilizado para depuración.

Pipes Personalizados

- Crear un nuevo Pipe es casi tan sencillo como crear una función a la que se pasa un valor y devuelve el valor transformado.
- Para crear un pipe se crea una clase decorada con @Pipe que implemente el interfaz PipeTransform, y se registra en el declarations del módulo. El método transform es el encargado de la función de transformación.

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'elipsis'
})
export class ElipsisPipe implements PipeTransform {
  transform(value: any, maxlen: number): any {
    return (!maxlen || !value || value.length < maxlen ||
        value.length < 4) ? value : (value.substr(0, maxlen -
        3) + '...');
  }
}
<div>{{ descripcion | elipsis:100}}</div>
```

Pipes impuros

- Angular busca cambios en los valores vinculados a los datos a través de un proceso de detección de cambios que se ejecuta después de cada evento DOM: pulsación de tecla, movimiento del mouse, tic del temporizador y respuesta del servidor. Esto podría ser costoso. Angular se esfuerza por reducir el costo siempre que sea posible y apropiado.
- Angular selecciona un algoritmo de detección de cambio más simple y más rápido cuando usa una pipe: solo cuando detecta un cambio puro en el valor de entrada, es decir, un cambio en un valor de entrada si es primitivo (String, Number, Boolean, Symbol) o una modificación en la referencia de objeto (Date, Array, Function, Object).
- Los pipes puros ignoran los cambios dentro de los objetos (compuestos). Los pipes impuros detectan los cambios durante cada ciclo de detección de cambio de componente.

```
@Pipe({
  pure: false
})
```

- La implementación de un pipe impuro requiere sumo cuidado dado que si la ejecución es costosa y de larga duración podría degradar la experiencia del usuario.

AsyncPipe

- El AsyncPipe (impuro) acepta un Promise o Observable como entrada y se suscribe automáticamente a la entrada y devuelve los valores asíncronamente cuando son emitidos.
- El AsyncPipe es con estado, mantiene una suscripción a la entrada Observable y sigue entregando valores del Observable medida que llegan.

```
<li *ngFor="let item of dao.list() | async">
```

FORMULARIOS

Formularios

- Un formulario crea una experiencia de entrada de datos coherente, eficaz y convincente.
- Un formulario Angular coordina un conjunto de controles de usuario enlazados a datos bidireccionalmente, hace el seguimiento de los cambios, valida la entrada y presenta errores.
- Angular ofrece dos tecnologías para trabajar con formularios: formularios basados en plantillas y formularios reactivos. Pero divergen marcadamente en filosofía, estilo de programación y técnica. Incluso tienen sus propios módulos: FormsModule y ReactiveFormsModule.
- Como el API de gestión de formularios cuenta con su propio módulo, es necesario importar y referenciar en el módulo principal:

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [ BrowserModule, FormsModule ],
  // …
})
export class AppModule { }
```

Formularios basados en plantillas

- Los formularios basados en plantillas delegan en la declaración de la plantilla la definición del formulario y el enlazado de datos por lo que requieren un código mínimo.
- Se colocan los controles de formulario HTML (<input>, <select>, <textarea>) en la plantilla del componente y se enlazan a las propiedades del modelo de datos suministrado por el componente, utilizando directivas como ngModel.
- No es necesario crear objetos de control de formulario Angular en la clase del componente dado que las directivas Angular los crean automáticamente, utilizando la información de los enlaces de datos.
- No es necesario traspasar los datos del modelo a los controles y viceversa dado que Angular lo automatiza con la directiva ngModel. Angular actualiza el modelo de datos mutable con los cambios de usuario a medida que suceden.
- Aunque esto significa menos código en la clase de componente, los formularios basados en plantillas son asíncronos, lo que puede complicar el desarrollo en escenarios más avanzados.

Encapsulación

- Angular encapsula automáticamente cada formulario y sus controles en un heredero de AbstractControl.
- Expone una serie de propiedades para gestionar los cambios, la validación y los errores:
  - pristine / dirty: Booleanos que indican si el formulario o control no ha sido o ya ha sido modificado por el usuario.
  - touched / untouched: Booleanos que indican si el formulario o control ha sido o no tocado por el usuario.
  - disabled / enabled: Booleanos que indican si el formulario o control esta deshabilitado o habilitado.
  - valid / invalid: Booleanos que indican si el formulario o control es valido o invalido.
  - pending : Booleano que indica si la validación está pendiente.
  - status: Cadena que indica el estado: VALID, INVALID, PENDING, DISABLED.
  - errors: Contiene como propiedades las validaciones que han fallado.
  - hasError() y getError() (v2.2): acceden a un error concreto.
  - value: valor del control

Avisos visuales

- Angular esta preparado para cambiar el estilo visual de los controles según su estado, utilizando class de CSS.
- Para personalizar el estilo (en la hojas de estilos):

```
.ng-valid[required], .ng-valid.required, .ng-pending {
  border-left: 5px solid #42A948; /* green */
}
.ng-invalid:not(form) {
  border-left: 5px solid #a94442; /* red */
}
```

- En caso de estar definidos Angular aplicará automáticamente los siguientes estilos:

Estado	Clase si es cierto	Clase si es falso
El control ha sido visitado	ng-touched	ng-untouched
El valor ha sido modificado	ng-dirty	ng-pristine
El valor es válido	ng-valid	ng-invalid

Directivas

- **ngForm**: Permite crear variables referencia al formulario.

```
<form name="miForm" #miDOMForm>
<form name="miForm" #miForm="ngForm">
        <button type="submit" [disabled]="miForm.invalid">
```

- **ngModel**: Permite el enlazado bidireccional y la creación de variables referencia.

```
<input type="text" name="nombre" required [(ngModel)]="model.nombre"
        #nombre="ngModel">
<div [hidden]="nombre.valid">Obligatorio</div>
<input type="text" #miDOMElement>
```

- **ngModelGroup**: Permite crear variables referencia de un grupo de controles de formulario.

```
<fieldset ngModelGroup="direccion" #direccionCtrl="ngModelGroup">
    <input name="calle" [ngModel]="direccion.calle" minlength="2">
    <input name="localidad" [ngModel]="direccion.localidad" required>
</fieldset >
```

Tipos de Validaciones

- required: el campo es requerido, aplicable a cualquier <input> , <select> o <textarea>, con la directiva required o [required] (vincula al modelo).

```
<input type="text" [(ngModel)]="model.nombre"
  name="nombre" required >
<input type="text" [(ngModel)]="model.nombre"
  name="nombre" [required]="reqNombre" >
```

- minlength: El campo debe tener un nº mínimo de caracteres.
- maxlength: El campo debe tener un nº máximo de caracteres.
- pattern: El campo debe seguir una expresión regular, applicable a <input> o a <textarea>.

```
<input type="text" [(ngModel)]="model.nombre"
  name="nombre" minlength="3" maxlength="50"
  pattern="/^[a-zA-Z]*$/" >
```


Tipos de Validaciones (v.2+)

- min: El campo debe tener un valor mínimo
- max: El campo debe tener un valor máximo

```
<input type="number"
  [(ngModel)]="model.edad" name="edad"
  min="18" max="99" >
```

- email: El campo debe tener el formato de un correo electrónico. Deberemos indicar en el <input> que type="email".

```
<input type="email"
  [(ngModel)]="model.correo" name="correo">
```

Comprobando las validaciones

- valid y invalid indican a nivel global si pasa la validación. Para saber que validaciones han fallado, la colección errors dispone de una propiedad por cada tipo de validación.

```
nombre.errors?.required
nombre.errors?.pattern
```

- Si no hay validaciones o si se cumplen todas las validaciones, la propiedad errors es nula por lo que es necesario utilizar el operador de navegación segura (?.) para evitar problemas
- Las propiedades del objeto errors se denominan de la misma forma que la validación que ha fallado y suministran información complementaria.
- La comprobación se puede realizar a nivel de control individual o de formulario completo.
- V 2.2: hasError y getError simplifican el acceso a los errores.
- Con el atributo HTML novalidate en la etiqueta <form> se evita que el navegador haga sus propias validaciones que choquen con las de Angular.

```
<form name="miForm" novalidate >
```


Notificación de errores

- Con la directiva *ngIf y con [hidden] se puede controlar cuando se muestran los mensajes de error.

```
<span class="error"
   [hidden]="!nombre?.errors?.maxlength">El tamaño
   m&aacute;ximo debe ser 50</span>
<span class="error"
   [hidden]="!nombre.hasError('required')">Es
   obligatorio</span>
```

- Para resumir en un mensaje varias causas de error:

```
<span class="error"
   [hidden]="!nombre?.errors?.minlength &&
   !nombre?.errors?.maxlength">El nombre debe tener
   entre 3 y 50 letras</span>
```

- Para no mostrar los mensajes desde el principio sin dar oportunidad al usuario a introducir los datos:

```
<span [hidden]="nombre.valid || miForm.pristine" …
```

Enviar formulario

- La propiedad disabled permite deshabilitar el botón de envío del formulario mientras no sea valido.

```
<button (click)="enviar()"
   [disabled]="miForm.invalid">Enviar</button>
```

- La directiva ngSubmit permite interceptar el evento de envío del formulario:

```
<form #miForm="ngForm" (ngSubmit)="onSubmit(miForm)">
    <input type="submit"
       [disabled]="miForm.invalid">Enviar</button>
onSubmit(miForm: NgForm) {
  if (miForm.valid) {
    …
  } else {
    alert("Hay datos inválidos");
  }
}
```


Consultas a la plantilla

- Los formularios basados en plantillas delegan la creación de sus controles de formulario y el enlazado en directivas, por lo que requieren un mínimo de código y acoplamiento entre plantilla y clase del componente.
- Se puede utilizar @ViewChild para obtener la primera etiqueta o directiva que coincida con el selector en la vista DOM. Si el DOM de la vista cambia y un nuevo hijo coincide con el selector, la propiedad se actualizará.
- Dado que los formularios basados en plantillas son asíncronos hay que controlar el ciclo de vida para evitar errores:

```
miForm: NgForm;
@ViewChild('miForm') currentForm: NgForm;

ngAfterViewChecked() {
 this.formChanged();
}
formChanged() {
 if (this.currentForm === this.miForm) { return; }
 this.miForm = this.currentForm;
 // ...
}
```

Validaciones personalizadas

```
import { Directive } from '@angular/core';
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';

@Directive({
 selector: '[upperCase]',
 providers: [{provide: NG_VALIDATORS, useExisting: UpperCaseValidatorDirective, multi: true}]
})
export class UpperCaseValidatorDirective implements Validator {
 validate(control: AbstractControl): {[key: string]: any} {
  const valor = control.value;
  if (valor) {
   return valor !== valor.toUpperCase() ? {'upperCase': {valor}} : null;
  } else {
   return null;
  }
 }
}
<input type="text" name="nombre" [(ngModel)]="model.nombre" #nombre="ngModel" upperCase>
<div *ngIf="nombre?.errors?.upperCase">Tiene que ir en mayúsculas.</div>
```


Validaciones personalizadas

```
import { Directive, forwardRef, Attribute } from '@angular/core';
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';

@Directive({
  selector: '[validateEqual][formControlName],[validateEqual][formControl],[validateEqual][ngModel]',
  providers: [ { provide: NG_VALIDATORS, useExisting: forwardRef(() => EqualValidatorDirective), multi: true } ]
})
export class EqualValidatorDirective implements Validator {
  constructor( @Attribute('validateEqual') public validateEqual: string) {}
  validate(control: AbstractControl): { [key: string]: any } {
    let valor = control.value;
    let cntrlBind = control.root.get(this.validateEqual);

    if (valor) {
      return (!cntrlBind || valor !== cntrlBind.value) ? { 'validateEqual': `${valor} <> ${cntrlBind.value}` } : null;
    }
    return null;
  }
}
<input type="password" ... #pwd="ngModel">
<input type="password" name="valPwd" id="valPwd" [(ngModel)]="valPwd" #valPwd="ngModel"
       validateEqual="pwd">
<div *ngIf="valPwd?.errors?. validateEqual">Tiene que ir en mayúsculas.</div>
```

Formularios Reactivos

- Las Formularios Reactivos facilitan un estilo reactivo de programación que favorece la gestión explícita de los datos que fluyen entre un modelo de datos (normalmente recuperado de un servidor) y un modelo de formulario orientado al UI que conserva los estados y valores de los controles HTML en la pantalla . Los formularios reactivos ofrecen la facilidad de usar patrones reactivos, pruebas y validación.
- Los formularios reactivos permiten crear un árbol de objetos de control de formulario Angular (AbstractControl) en la clase de componente y vincularlos a elementos de control de formulario nativos (DOM) en la plantilla de componente.
- Así mismo, permite crear y manipular objetos de control de formulario directamente en la clase de componente. Como la clase del componente tiene acceso inmediato tanto al modelo de datos como a la estructura de control de formulario, puede trasladar los valores del modelo de datos a los controles de formulario y retirar los valores modificados por el usuario. El componente puede observar los cambios en el estado del control de formulario y reaccionar a esos cambios.

Formularios Reactivos

- Una ventaja de trabajar directamente con objetos de control de formulario es que las actualizaciones de valores y las validaciones siempre son sincrónicas y bajo tu control. No encontrarás los problemas de tiempo que a veces plaga los formularios basados en plantillas.
- Las pruebas unitaria pueden ser más fáciles en los formularios reactivos al estar dirigidos por código.
- De acuerdo con el paradigma reactivo, el componente preserva la inmutabilidad del modelo de datos, tratándolo como una fuente pura de valores originales. En lugar de actualizar el modelo de datos directamente, la clase del componente extrae los cambios del usuario y los envía a un componente o servicio externo, que hace algo con ellos (como guardarlos) y devuelve un nuevo modelo de datos al componente que refleja el estado del modelo actualizado en el formulario.
- El uso de las directivas de formularios reactivos no requiere que se sigan todos los principios reactivos, pero facilita el enfoque de programación reactiva si decide utilizarlo.

Formularios Reactivos

- Como el API de gestión de formularios cuenta con su propio módulo, es necesario importar y referenciar en el módulo principal:

```
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  imports: [ BrowserModule, ReactiveFormsModule ],
  // …
})
export class AppModule { }
```

- Se pueden usar los dos paradigmas de formularios en la misma aplicación.
- La directiva ngModel no forma parte del ReactiveFormsModule, para poder utilizarla hay importar FormsModule y, dentro de un formulario reactivo, hay que acompañarla con:

```
[ngModelOptions]="{standalone: true}"
```


Clases esenciales

- **AbstractControl**: es la clase base abstracta para las tres clases concretas de control de formulario: FormControl, FormGroup y FormArray. Proporciona sus comportamientos y propiedades comunes, algunos de los cuales son observables.
- **FormControl**: rastrea el valor y el estado de validez de un control de formulario individual. Corresponde a un control de formulario HTML, como un <input>, <select> o <textarea>.
- **FormGroup**: rastrea el valor y el estado de validez de un grupo de instancias de AbstractControl. Las propiedades del grupo incluyen sus controles hijos. El formulario en si es un FormGroup .
- **FormArray**: rastrea el valor y el estado de validez de una matriz indexada numéricamente de instancias de AbstractControl.
- **FormBuilder**: es un servicio que ayuda a generar un FormGroup mediante una estructura que reduce la complejidad, la repetición y el desorden al crear formularios.

Creación del formulario

- Hay que definir en la clase del componente una propiedad publica de tipo FormGroup a la que se enlazará la plantilla.
- Para el modelo:

```
model = { user: 'usuario', password: 'P@$$w0rd', roles: [{
        role: 'Admin' }, { role: 'User' }] };
```

- Hay que definir los diferentes FormControl a los que se enlazaran los controles de la plantilla instanciándolos con el valor inicial y, opcionalmente, las validaciones:

```
let pwd = new FormControl('P@$$w0rd',
        Validators.minLength(2));
```


Creación del formulario

- Instanciar el FormGroup pasando un array asociativo con el nombre del control con la instancia de FormControl y, opcionalmente, las validaciones conjuntas:

```
const form = new FormGroup({
  password: new FormControl('', Validators.minLength(2)),
  passwordConfirm: new FormControl('',
        Validators.minLength(2)),
}, passwordMatchValidator);
```

- El FormGroup es una colección de AbstractControl que se puede gestionar dinámicamente con los métodos addControl, setControl y removeControl.

Sub formularios

- Un FormGroup puede contener otros FormGroup (sub formularios), que se pueden gestionar, validar y asignar individualmente:

```
const form = new FormGroup({
  user: new FormControl(''),
  password: new FormGroup({
     passwordValue: new FormControl('',
          Validators.minLength(2)),
     passwordConfirm: new FormControl('',
          Validators.minLength(2)),
   }, passwordMatchValidator);
});
```


Agregaciones

- Un FormGroup puede contener uno o varios arrays (indexados numéricamente) de FormControl o FormGroup para gestionar las relaciones 1 a N.
- Hay que crear una instancia de FormArray y agregar un FormGroup por cada uno de los N elementos.

```
const fa = new FormArray();
this.model.roles.forEach(r => fa.push(
  new FormGroup({ role: new FormControl(r.role , Validators.required) })
));
this.miForm = new FormGroup({
  user: new FormControl(''),
  password: // ...
  roles: fa
});
```

Agregaciones

- Añadir un nuevo elemento a la agregación:

```
addRole() {
  (this.miForm.get('roles') as FormArray).push(
       new FormGroup({ role: new FormControl(r.role ,
           Validators.required) })
  );
}
```

- Borrar un elemento de la agregación:

```
deleteRole(ind: number) {
  (this.miForm.get('roles') as FormArray).removeAt(ind);
}
```


Validaciones

- La clase Validators expone las validaciones predefinidas en Angular y se pueden crear funciones de validación propias.
- Las validaciones se establecen cuando se instancian los FormControl o FormGroup. En caso de múltiples validaciones se suministra una array de validaciones:

```
this.miForm = new FormGroup({
  user: new FormControl('', [Validators.required,
            Validators.minLength(2), Validators.maxLength(20)]),
  password: new FormGroup({…}, passwordMatchValidator);
  roles: fa
});
```

Validaciones personalizadas

- Para un FormControl (por convenio: sin valor se considera valido salvo en required):

```
export function naturalNumberValidator(): ValidatorFn {
   return (control: AbstractControl): { [key: string]: any } => {
      return (!control.value || /^[1-9]\d*$/.test(control.value)) ?
            null : { naturalNumber: { valid: false } };
   };
}
```

- Para un FormGroup:

```
function passwordMatchValidator(g: FormGroup) {
  return g.get('passwordValue').value ===
            g.get('passwordConfirm').value ? null : {'mismatch': true};
}
```


Enlace de datos

- Según el paradigma reactivo, el componente preserva la inmutabilidad del modelo de datos, por lo que es necesario traspasar manualmente los datos del modelo a los controles y viceversa.
- Con los setValue y patchValue se pasan los datos hacia el formulario. El método setValue comprueba minuciosamente el objeto de datos antes de asignar cualquier valor de control de formulario. No aceptará un objeto de datos que no coincida con la estructura FormGroup o falte valores para cualquier control en el grupo. patchValue tiene más flexibilidad para hacer frente a datos divergentes y fallará en silencio.

```
this.miForm.setValue(this.modelo);
this.miForm.get('user').setValue(this.modelo.user);
```

- Para recuperar los datos del formulario se dispone de la propiedad value:

```
aux = this.miForm.value;
this.modelo.user = this.miForm.get('user').value;
```

- Para borrar los datos del formulario o control:

```
this.miForm.reset();
```

Detección de cambios

- Los herederos de AbstractControl exponen Observables, a través de valueChanges y statusChanges, que permiten suscripciones que detectan los cambios en el formulario y sus controles.
- Para un formulario o FormGroup:

```
this.miForm.valueChanges
  .subscribe(data => {
    // data: datos actuales del formulario
  });
```

- Para un FormControl:

```
this.miForm.get('user').valueChanges
  .subscribe(data => {
    // data: datos actuales del control
  });
```


FormBuilder

- El servicio FormBuilder facilita la creación de los formularios. Es necesario inyectarlo.

```
constructor(protected fb: FormBuilder) {}
```

- Mediante una estructura se crea el formulario:

```
const fa = this.fb.array([]);
this.model.roles.forEach(r => fa.push(this.fb.group({ role: [r.role,
            Validators.required] })));
this.miForm = this.fb.group({
  user: [this.model.user, [Validators.required, Validators.minLength(2),
            Validators.maxLength(20)]],
  password: this.fb.group({
     passwordValue: [this.model.password, Validators.minLength(2)],
     passwordConfirm: '',
   }, { validator: passwordMatchValidator }),
  roles: fa
});
```

Directivas

- formGroup: para vincular la propiedad formulario al formulario.

```
<form [formGroup]="miForm" >
```

- formGroupName: establece una etiqueta contenedora para un subformulario (FormGroup dentro de otro FormGroup).

```
<div formGroupName="password" >
```

- formControlName: para vincular <input>, <select> o <textarea> a su correspondiente FormControl. Debe estar correctamente anidado dentro de su formGroup o formControlName.

```
<input type="password" formControlName="passwordValue" >
```

- formArrayName: establece una etiqueta contenedora para un array de FormGroup. Se crea un formGroupName con el valor del índice del array.

```
<div formArrayName="roles">
    <div *ngFor="let row of elementoForm.get('roles').controls; let i=index"
          [formGroupName]="i" >
       <input type="text" formControlName="role" >
</div></div>
```


Mostrar errores

- Errores en los controles del formulario:

```
<!-- null || true -->
<span class="errorMsg" [hidden]="!miForm?.controls['user'].errors?.required">Es obligatorio.</span>
<span class="errorMsg" [hidden]="!miForm?.get('user')?.errors?.required">Es obligatorio.</span>
<!-- false || true -->
<span class="errorMsg" [hidden]="!miForm.hasError('required', 'user')">Es obligatorio.</span>
```

- Errores en los controles de sub formularios:

```
{{miForm?.get('password')?.get('passwordValue')?.errors | json}}
{{miForm?.get(['password', 'passwordValue'])?.errors | json}}
```

- Errores en las validaciones del FormGroup:

```
{{miForm?.get('password')?.errors | json}}
```

- Errores en los elementos de un FormArray:

```
<ul *ngFor="let row of miForm.get('roles').controls … >
   {{row?.get('role')?.errors | json}}
```

Plantilla (I)

```
<form [formGroup]="miForm">
  <label>User: <input type="text" formControlName="user"
       ></label>
  {{miForm?.get('user')?.errors | json}}
  <fieldset formGroupName="password" >
   <label>Password: <input type="password"
       formControlName="passwordValue" ></label>
   {{miForm?.get('password')?.get('passwordValue')?.errors |
       json}}
   <label>Confirm Password: <input type="password"
       formControlName="passwordConfirm" ></label>
  </fieldset>
  {{miForm?.get('password')?.errors | json}}
```


Plantilla (II)

```
  <div formArrayName="roles">
   <h4>Roles</h4><button (click)="addRole()">Add
       Role</button>
   <ul *ngFor="let row of $any(miForm.get('roles')).controls;
       let i=index" [formGroupName]="i">
    <li>{{i + 1}}: <input type="text" formControlName="role">
    {{row?.get('role')?.errors | json}}
    <button (click)="deleteRole(i)">Delete</button></li>
   </ul>
  </div>
  <button (click)="send()">Send</button>
</form>{{ miForm.value | json }}
```

Sincronismo

- Los formularios reactivos son síncronos y los formularios basados en plantillas son asíncronos.
- En los formularios reactivos, se crea el árbol de control de formulario completo en el código. Se puede actualizar inmediatamente un valor o profundizar a través de los descendientes del formulario principal porque todos los controles están siempre disponibles.
- Los formularios basados en plantillas delegan la creación de sus controles de formulario en directivas. Para evitar errores "changed after checked", estas directivas tardan más de un ciclo en construir todo el árbol de control. Esto significa que debe esperar una señal antes de manipular cualquiera de los controles de la clase de componente.
- Por ejemplo, si inyecta el control de formulario con una petición @ViewChild (NgForm) y se examina en el ngAfterViewInit, no tendrá hijos. Se tendrá que esperar, utilizando setTimeout, antes de extraer un valor de un control, validar o establecerle un nuevo valor.
- La asincronía de los formularios basados en plantillas también complica las pruebas unitarias. Se debe colocar el bloque de prueba en async() o en fakeAsync() para evitar buscar valores en el formulario que aún no están disponibles. Con los formularios reactivo, todo está disponible tal y como se espera que sea.

JavaScript Object Notation

http://tools.ietf.org/html/rfc4627

JSON

Introducción

- JSON (JavaScript Object Notation) es un formato sencillo para el intercambio de información.
- El formato JSON permite representar estructuras de datos (arrays) y objetos (arrays asociativos) en forma de texto.
- La notación de objetos mediante JSON es una de las características principales de JavaScript y es un mecanismo definido en los fundamentos básicos del lenguaje.
- En los últimos años, JSON se ha convertido en una alternativa al formato XML, ya que es más fácil de leer y escribir, además de ser mucho más conciso.
- No obstante, XML es superior técnicamente porque es un lenguaje de marcado, mientras que JSON es simplemente un formato para intercambiar datos.
- La especificación completa del JSON es la RFC 4627, su tipo MIME oficial es application/json y la extensión recomendada es .json.

Estructuras

- JSON está constituido por dos estructuras:
  - Una colección de pares de nombre/valor. En los lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
  - Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como tablas, arreglos, vectores, listas o secuencias.
- Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

Sintaxis

- Un array es un conjunto de valores separados por comas (,) que se encierran entre corchetes […]
- Un objeto es un conjunto de pares nombre:valor separados por comas (,) que se acotan entre llaves { … }
- Los nombre son cadenas, entre comillas dobles (").
- El separador entre el nombre y el valor son los dos puntos (:)
- El valor debe ser un objeto, un array, un número, una cadena o uno de los tres nombres literales siguientes (en minúsculas):
  - true, false o null
- Se codifica en Unicode, la codificación predeterminada es UTF-8.

Valores numéricos

- La representación de números es similar a la utilizada en la mayoría de los lenguajes de programación.
- Un número contiene una parte entera que puede ser prefijada con un signo menos opcional, que puede ser seguida por una parte fraccionaria y / o una parte exponencial.
- La parte fraccionaria comienza con un punto (como separador decimal) seguido de uno o más dígitos.
- La parte exponencial comienza con la letra E en mayúsculas o minúsculas, lo que puede ser seguido por un signo más o menos, y son seguidas por uno o más dígitos.
- Los formatos octales y hexadecimales no están permitidos. Los ceros iniciales no están permitidos.
- No se permiten valores numéricos que no se puedan representar como secuencias de dígitos (como infinito y NaN).

Valores cadena

- La representación de las cadenas es similar a las convenciones utilizadas en la familia C de lenguajes de programación.
- Una cadena comienza y termina con comillas (").
- Se pueden utilizar todos los caracteres Unicode dentro de las comillas con excepción de los caracteres que se deben escapar: los caracteres de control (U + 0000 a U + 001F) y los caracteres con significado.
- Cuando un carácter se encuentra fuera del plano multilingüe básico (U + 0000 a U + FFFF), puede ser representado por su correspondiente valor hexadecimal. Las letras hexadecimales A-F puede ir en mayúsculas o en minúsculas.
- Secuencias de escape:
  - \\, \/, \", \n, \r, \b, \f, \t
  - \u[0-9A-Fa-f]{4}

Objeto con anidamientos

```
{
     "Image": {
         "Width":  800,
         "Height": 600,
         "Title":  "View from 15th Floor",
         "Thumbnail": {
             "Url":    "/image/481989943",
             "Height": 125,
             "Width":  "100"
             },
         "IDs": [116, 943, 234, 38793]
         }
 }
```

Array de objetos

```
[
    {
       "precision": "zip",
       "Latitude":  37.7668,
       "Longitude": -122.3959,
       "City":      "SAN FRANCISCO",
       "State":     "CA",
       "Zip":       "94107"
    },
    {
       "precision": "zip",
       "Latitude":  37.371991,
       "Longitude": -122.026020,
       "City":      "SUNNYVALE",
       "State":     "CA",
       "Zip":       "94085"
    }
]
```


JSON en JavaScript

- El Standard Built-in ECMAScript Objects define que todo interprete de JavaScript debe contar con un objeto JSON como miembro del objeto Global.
- El objeto debe contener, al menos, los siguientes miembros:
  - **JSON.parse** (Función): Convierte una cadena de la notación de objetos de JavaScript (JSON) en un objeto de JavaScript.
  - **JSON.stringify** (Función): Convierte un valor de JavaScript en una cadena de la notación de objetos JavaScript (JSON).

REpresentational State Transfer

SERVICIOS RESTFUL

REST (REpresentational State Transfer)

- Un **estilo de arquitectura** para desarrollar aplicaciones web distribuidas que se basa en el uso del protocolo HTTP e Hypermedia.
- Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba.
- El HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado:
  - HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
- El HTTP permite en el encabezado transmitir la información de comportamiento:
  - Accept, Content-type, Response (códigos de estado), Authorization, Cache-control, …

Objetivos de los servicios REST

- Desacoplar el cliente del backend
- Mayor escalabilidad
  - Sin estado en el backend.
- Separación de problemas
- División de responsabilidades
- API uniforme para todos los clientes
  - Disponer de una interfaz uniforme (basada en URIs)

Uso de la cabecera

- **Request**: Método /uri?parámetros
  - GET: Recupera el recurso
    - Todos: Sin parámetros
    - Uno: Con parámetros
  - POST: Crea un nuevo recurso
  - PUT: Edita el recurso
  - DELETE: Elimina el recurso
- **Accept**: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
- **Content-type**: Indica en que formato está codificado el cuerpo, utilizando MIME
- **Response**: Código de estado con el que el servidor informa del resultado de la petición.

Peticiones

Request: GET /users
    Response: 200
    content-type:application/json
Request: GET /users/11
    Response: 200
    content-type:application/json
Request: POST /users
    Response: 201 Created
    content-type:application/json
    body
Request: PUT /users/11
    Response: 200
    content-type:application/json
    body
Request: DELETE /users/11
    Response: 204 No Content

Fake Online REST API
for Testing and Prototyping
https://jsonplaceholder.typicode.com/

Richardson Maturity Model

http://www.crummy.com/writing/speaking/2008-QCon/act3.html

- # Nivel 1 (Pobre): Se usan URIs para identificar recursos:
  - Se debe identificar un recurso
    /invoices/?page=2 → /invoices/page/2
  - Se construyen con nombres nunca con verbos
    /getUser/{id} → /users/{id}/
    /users/{id}/edit/login → users/{id}/access-token
  - Deberían tener una estructura jerárquica
    /invoices/user/{id} → /user/{id}/invoices
- # Nivel 2 (Medio): Se usa el protocolo HTTP adecuadamente
- # Nivel 3 (Óptimo): Se implementa hypermedia.

Hypermedia

- Se basa en la idea de enlazar recursos: propiedades que son enlaces a otros recursos.
- Para que sea útil, el cliente debe saber que en la respuesta hay contenido hypermedia.
- En content-type es clave para esto
  - Un tipo genérico no aporta nada:
    Content-Type: text/xml
  - Se pueden crear tipos propios
    Content-Type:application/servicio+xml

JSON Hypertext Application Language

- RFC4627 http://tools.ietf.org/html/draft-kelly-json-hal-00
- Content-Type: application/hal+json

```
{
    "_links": {
      "self": {"href": "/orders/523" },
      "warehouse": {"href": "/warehouse/56" },
      "invoice": {"href": "/invoices/873“}
    },
    , "currency": "USD"
    , "status": "shipped"
    , "total": 10.20
}
```

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de objetos asociados que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
  - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
  - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

GraphQL

http://graphql.org/

- GraphQL es un lenguaje para ejecutar consultas en un API de servidor mediante el cual se provee al cliente de una descripción detallada de las estructuras de datos que ofrece.
- Este lenguaje fue creado por Facebook en 2012, cuya estandarización se comenzó a desarrollar en 2015, por lo que es un lenguaje en continuo desarrollo.
- Las principales características y funcionalidades que ofrece GraphQL son las siguientes:
  - Describe la estructura/organización de los datos ofrecidos por el API mediante un sistema de fuertemente tipado en el que se detallan los recursos y las relaciones existentes entre estos.
  - Permite solicitar al servidor tan sólo la información que necesitamos de un recurso en una sola petición incluyendo los datos de otros recursos que se relacionen con el recurso raíz.
  - Ofrece una única URI en la que se realizan todas las peticiones, por lo que no debemos preocuparnos de la semántica implementada por cada API.
  - Es un lenguaje independiente de la fuente de datos. Los datos pueden obtenerse de un servicio web, una base de datos, un fichero, etc.
  - Permite la evolución de un API sin crear nuevas versiones
  - Existen múltiples librerías para implementar en distintos lenguajes (JavaScript, .NET, Ruby, Java …)

GraphQL

Definición de datos

```
type Human implements Character {
 id: ID!
 name: String!
 friends: [Character]
 appearsIn: [Episode]!
 starships: [Starship]
 totalCredits: Int
}
enum Episode {
 NEWHOPE
 EMPIRE
 JEDI
}
type Starship {
 id: ID!
 name: String!
 length(unit: LengthUnit = METER): Float
}
```

Consulta

```
{
 human(id: 1002) {
  name
  appearsIn
  starships {
   name
  }
 }
}
```


GraphQL

Resultado

```
{
 "data": {
  "human": {
   "name": "Han Solo",
   "appearsIn": [
    "NEWHOPE",
    "EMPIRE",
    "JEDI"
   ],
   "starships": [
    {
     "name": "Millenium Falcon"
    },
    {
     "name": "Imperial shuttle"
    }
   ]
  }
 }
}
```

ACCESO AL SERVIDOR

RxJS

- La Programación Reactiva (o Reactive Programming) es un paradigma de programación asíncrono relacionado con los flujos de datos y la propagación de los cambios. Estos flujos se pueden observar y reaccionar en consecuencia.
- Reactive Extension for JavaScript (RxJS) es una biblioteca para implementar programación reactiva utilizando observables que facilita la composición de flujos de datos asíncronos o basados en eventos.
- Un Observable es un mecanismo creado para representar los flujos de datos que, cada vez que cambian los datos, informa de los cambios producidos. Los Observables se basan en dos patrones de programación bien conocidos como son el patrón “Observer” y el patrón “Iterator”.
- Un Observador o suscriptor esta interesado en enterarse de los cambios que se producen en un flujo de datos.
- Los observadores se subscriben a los Observables para recibir notificaciones de los cambios. Un Observable puede tener tantos suscriptores como sean necesario.

RxJS

- Para hacerlo simple, podríamos decir que un Observable es un objeto que guarda un valor y que emite un evento a todos sus suscriptores cada vez que ese valor se actualiza.
- En palabras de la RxJS, un Observable es un conjunto de valores a lo largo de cualquier intervalo de tiempo.
- La librería RxJS aporta un amplio conjunto de recursos para la creación de Observables sobre diversas fuentes de datos, además de varios operadores para transformar los resultados de los Observables como:
  - el operador map (equivalente al método map de los arrays JS para manipular cada elemento del array),
  - el operador debounce para ignorar eventos demasiado seguidos,
  - el operador merge para combinar los eventos de 2 o más observables en uno…

  https://github.com/Reactive-Extensions/RxJS

Servicio HttpClient (v 4.3)

- El servicio HttpClient permite hacer peticiones AJAX al servidor.
- Encapsula el objeto XMLHttpRequest (Level 2), pero está integrado con Angular como un servicio (con todas las ventajas de ellos conlleva) y notifica a Angular que ha habido un cambio en el modelo de JavaScript y actualiza la vista y el resto de dependencias adecuadamente.
- El HttpClient es un servicio opcional y no es parte del núcleo Angular. Es parte de la biblioteca @angular/common/http, con su propio módulo.
- Hay que importar lo que se necesita de él en el módulo principal como se haría con cualquier otro modulo Angular. No es necesario registrar el proveedor.

```
import { HttpClientModule, HttpClientJsonpModule } from
    '@angular/common/http';
@NgModule({
  imports: [ HttpModule, HttpClientJsonpModule, // …
  ], // …
})
export class AppModule { }
```

HttpClient (v 4.3)

- Evolución del Http anterior:
  - Atajos a los verbos HTTP con parámetros mínimos.
  - Respuestas tipadas.
  - Acceso al cuerpo de respuesta síncrono y automatizado, incluido el soporte para cuerpos de tipo JSON.
  - JSON como valor predeterminado, ya no necesita ser analizado explícitamente.
  - Los interceptores permiten que la lógica de middleware sea insertada en el flujo.
  - Objetos de petición/respuesta inmutables
  - Eventos de progreso para la carga y descarga de la solicitud.
  - Protección ante XSRF

Servicio HttpClient

- Método general:

  request(first: string|HttpRequest<any>, url?: string, options: {...}): Observable<any>
- Atajos:

  get(url: string, options: {...}): Observable<any>

  post(url: string, body: any, options: {...}): Observable<any>

  put(url: string, body: any, options: {...}): Observable<any>

  delete(url: string, options: {...}): Observable<any>

  patch(url: string, body: any, options: {...}): Observable<any>

  head(url: string, options: {...}): Observable<any>

  options(url: string, options: {...}): Observable<any>

  jsonp<T>(url: string, callbackParam: string): Observable<T>

Opciones adicionales

- **body**: Para peticiones POST o PUT
- **headers**: Colección de cabeceras que acompañan la solicitud
- **observe**: **'body'** | 'events' | 'response'
- **params**: Coleccion de pares nombre/valor del QueryString
- **reportProgress**: true activa el seguimiento de eventos de progreso
- **responseType**: 'arraybuffer'|'blob'|**'json'**|'text',
- **withCredentials**: true indica el envío de credenciales

Peticiones al servidor

- Solicitar datos al servidor:

```
this.http.get(`ws/entidad/${Id} `)
    .subscribe(
      datos => this.listado = datos,
      error => console.error(`Error: ${error}`)
    );
```

- Enviar datos al servidor (ws/entidad/edit?id=3):

```
this.http.post('ws/entidad', body, {
  headers: new HttpHeaders().set('Authorization', 'my-auth-token'),
  params: new HttpParams().set('id', '3'),
})
.subscribe(
  data => console.log('Success uploading', data),
  error => console.error('Error: ${error}')
);
```

JSON como valor predeterminado

- El tipo más común de solicitud para un backend es en formato JSON.
  return this.http.get(this.baseUrl); // .map(response => response.json());
- Se puede establecer el tipo de datos esperado en la respuesta:
  this.http.get<MyModel>(this.baseUrl).subscribe(data => {
  // data is an instance of type MyModel
  });
- Para obtener la respuesta completa:
  this.http.get<MyModel>(this.baseUrl, {observe: 'response'})
  .subscribe(resp => {
  // resp is an instance of type HttpResponse
  // resp.body is an instance of type MyModel
  });
- Para solicitar datos que no sean JSON:
  return this.http.get(this.baseUrl, {responseType: 'text'});

HttpResponse<T>

- **type**: HttpEventType.Response o HttpEventType.ResponseHeader.
- **ok**: Verdadero si el estado de la respuesta está entre 200 y 299.
- **url**: URL de la respuesta.
- **status**: Código de estado devuelto por el servidor.
- **statusText**: Versión textual del `status`.
- **headers**: Cabeceras de la respuesta.
- **body**: Cuerpo de la respuesta en el formato establecido.

Manejo de errores

- Hay que agregar un controlador de errores a la llamada de .subscribe():

```
this.http.get<MyModel>(this.baseUrl).subscribe(
  resp => { ... },
  err => { console.log('Error !!!');  }
 });
```

- Para obtener los detalles del error:

```
(err: HttpErrorResponse) => {
    if (err.error instanceof Error) {
      // A client-side or network error occurred. Handle it accordingly.
      console.log('An error occurred:', err.error.message);
    } else {
      // The backend returned an unsuccessful response code.
      // The response body may contain clues as to what went wrong,
      console.log(`Backend returned code ${err.status}, body was: ${err.error}`);
    }
  }
```


HttpErrorResponse

- **type**: HttpEventType.Response o HttpEventType.ResponseHeader.
- **url**: URL de la respuesta.
- **status**: Código de estado devuelto por el servidor.
- **statusText**: Versión textual del `status`.
- **headers**: Cabeceras de la respuesta.
- **ok**: Falso
- **name**: 'HttpErrorResponse'
- **message**: Indica si el error se ha producido en la solicitud (status: 4xx, 5xx) o al procesar la respuesta (status: 2xx)
- **error**: Cuerpo de la respuesta.

Tratamiento de errores

- Una forma de tratar los errores es simplemente reintentar la solicitud.
- Esta estrategia puede ser útil cuando los errores son transitorios y es poco probable que se repitan.
- RxJS tiene el operador .retry(), que automáticamente resubscribe a un Observable, reeditando así la solicitud, al encontrarse con un error.

```
import {retry, catchError} from 'rxjs/internal/operators';
// …
this.http.get<MyModel>(this.baseUrl).pipe(
   retry(3),
   catchError(err => …)
   ).subscribe(…);
```


Retroceso exponencial

- El retroceso exponencial es una técnica en la que se vuelve a intentar una API después de la falla, lo que hace que el tiempo entre reintentos sea más prolongado después de cada fallo consecutiva, con un número máximo de reintentos después de los cuales se considera que la solicitud ha fallado.

```
function backoff(maxTries, ms) {
 return pipe(
   retryWhen(attempts => range(1, maxTries)
     .pipe(zip(attempts, (i) => i), map(i => i * i), mergeMap(i =>  timer(i * ms))
     ) ) );
}

ajax('/api/endpoint')
  .pipe(backoff(3, 250))
  .subscribe(data => handleData(data));
```

Servicio RESTFul

```
import { HttpClient } from '@angular/common/http';
import {Observable} from 'rxjs';
import { environment } from 'src/environments/environment';

export abstract class RESTDAOService<T, K> {
  protected baseUrl = environment.apiURL;
  constructor(protected http: HttpClient, entidad: string, protected option = {}) {
        this.baseUrl += entidad; }
  query(): Observable<T> { return this.http.get<T>(this.baseUrl, this.option); }
  get(id: K): Observable<T> { return this.http.get<T>(this.baseUrl + '/' + id, this.option);
        }
  add(item: T): Observable<T>  { return this.http.post<T>(this.baseUrl, item,
        this.option); }
  change(id: K, item: T): Observable<T> {return this.http.put<T>(this.baseUrl + '/' + id,
        item, this.option);}
  remove(id: K): Observable<T> { return this.http.delete<T>(this.baseUrl + '/' + id,
        this.option); }
}
```


Seguridad

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - **https**://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com**:8080**/scripts/codigo2.js
  - http://**scripts**.ejemplo.com/codigo2.js
  - http://**192.168.0.1**/scripts/codigo2.js
- La propiedad document.domain se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación.

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece y, por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.
- XMLHttpRequest sigue la política de mismo-origen, por lo que solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron a los proveedores de navegadores que permitieran a XMLHttpRequest realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE“
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

Para habilitar CORS en servidores específicos consultar http://enable-cors.org

Proxy a un servidor de back-end

- Se puede usar el soporte de proxy en el servidor de desarrollo del webpack para desviar ciertas URL a un servidor backend.
- Crea un archivo proxy.conf.json junto a angular.json.

```
{
  "/api": {
    "target": "http://localhost:4321",
    "pathRewrite": { "^/api": "/ws" },
    "secure": false,
    "logLevel": "debug"
  }
}
```

- Cambiar la configuración del Angular CLI en angular.json:

```
"architect": {
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
      ...
      "proxyConfig": "proxy.conf.json"
    },
```

Desactivar la seguridad de Chrome

- Pasos para Windows:
  - Localizar el acceso directo al navegador (icono) y crear una copia como “Chrome Desarrollo”.
  - Botón derecho -> Propiedades -> Destino
  - Editar el destino añadiendo el parámetro al final. ej: "C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-web-security
  - Aceptar el cambio y lanzar Chrome
- Para desactivar parcialmente la seguridad:

  --allow-file-access

  --allow-file-access-from-files

  --allow-cross-origin-auth-prompt
- Referencia a otros parametros:
  - http://peter.sh/experiments/chromium-command-line-switches/

JSONP (JSON con Padding)

- JSONP es una técnica de comunicación utilizada en los programas JavaScript para realizar llamadas asíncronas a dominios diferentes. JSONP es un método concebido para suplir la limitación de AJAX entre dominios por razones de seguridad. Esta restricción no se aplica a la etiqueta <script> de HTML, para la cual se puede especificar en su atributo src la URL de un script alojado en un servidor remoto.
- En esta técnica se devuelve un objeto JSON envuelto en la llamada de una función (debe ser código JavaScript valido), la función ya debe estar definida en el entorno de JavaScript y se encarga de manipular los datos JSON.

```
miJsonCallback ({"Nombre":"Carmelo","Apellidos":"Cotón"});
```

- Por convención, el nombre de la función de retorno se suele especificar mediante un parámetro de la consulta, normalmente, utilizando jsonp o callback como nombre del campo en la solicitud al servidor.

```
<script type="text/javascript"
   src="http://otrodominio.com/datos.json?callback=
   miJsonCallback"></script>
```

JSONP (JSON con Padding)

- Si no se importa el módulo HttpClientJsonpModule, las solicitudes de Jsonp llegarán al backend con el método JSONP, donde serán rechazadas.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class WikipediaService {
 constructor(private http: HttpClient) {}
 search (term: string) {
   let wikiUrl =
        `http://en.wikipedia.org/w/api.php?search=${term}&action=opensearch&for
        mat=json`;
   return this.http.jsonp(wikiUrl, `callback`).subscribe(
      datos => this.items  = datos, error => console.error(`Error: ${error}`)
    );
 }
}
```


Eventos

- Los eventos trabajan en un nivel más bajo que las solicitudes. Una sola solicitud puede generar múltiples eventos.
- Tipos de eventos:
  - **Sent**: La solicitud fue enviada.
  - **ResponseHeader**: Se recibieron el código de estado de respuesta y los encabezados.
  - **UploadProgress**: Se ha recibido un evento de progreso de subida.
  - **DownloadProgress**: Se ha recibido un evento de progreso de descarga.
  - **Response**: Se ha recibido la respuesta completa incluyendo el cuerpo.
  - **User**: Un evento personalizado de un interceptor o un backend.

Eventos de progreso

- A veces las aplicaciones necesitan transferir grandes cantidades de datos, como por ejemplo subir ficheros, y esas transferencias pueden tomar mucho tiempo.
- Es una buena práctica para la experiencia de usuario proporcionar información sobre el progreso de tales transferencias.

```
this.http.post('/upload/file', file, { reportProgress: true,  })
  .subscribe(event => {
    if (event.type === HttpEventType.UploadProgress) {
      const percentDone = Math.round(100 * event.loaded / event.total);
      console.log(`File is ${percentDone}% uploaded.`);
    } else if (event instanceof HttpResponse) {
      console.log('File is completely uploaded!');
    }
  });
```


Interceptores

- Una característica importante de HttpClient es la interceptación: la capacidad de declarar interceptores que se sitúan entre la aplicación y el backend.
- Cuando la aplicación hace una petición, los interceptores la transforman antes de enviarla al servidor.
- Los interceptores pueden transformar la respuesta en su camino de regreso antes de que la aplicación la vea.
- Esto es útil para múltiples escenarios, desde la autenticación hasta el registro.
- Cuando hay varios interceptores en una aplicación, Angular los aplica en el orden en que se registraron.

Crear un interceptor

- Los interceptores son servicios que implementan el interfaz HttpInterceptor, que requiere el método intercept:

```
intercept(req: HttpRequest<any>, next: HttpHandler):
        Observable<HttpEvent<any>> {
    return next.handle(req);
}
```

- next siempre representa el siguiente interceptor en la cadena, si es que existe, o el backend final si no hay más interceptores.
- La solicitud es inmutable para asegurar que los interceptores vean la misma petición para cada reintento. Para modificarla es necesario crear una nueva con el método clone():

```
return next.handle(req.clone({url: req.url.replace('http://', 'https://')}));
```

- Se registran como un servicio múltiple sobre HTTP_INTERCEPTORS en el orden deseado:

```
{ provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true, },
```


Modificar la petición

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

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private auth: AuthService) { }

  intercept(req: HttpRequest<any>, next: HttpHandler):
        Observable<HttpEvent<any>> {
    if (!req.withCredentials || ! this.auth.isAutenticated)
        { return next.handle(req); }
    const authReq = req.clone({ headers: req.headers.set('Authorization',
        this.auth.AuthorizationHeader) });
    return next.handle(authReq);
  }
}
```

Modificar la respuesta

```
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest, HttpResponse } from
          '@angular/common/http';
import { tap, finalize } from 'rxjs/operators';
@Injectable()
export class LoggingInterceptor implements HttpInterceptor {
 intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const started = Date.now();
    let ok: string;
    return next.handle(req)
      .pipe(
        tap(
          event => ok = event instanceof HttpResponse ? 'succeeded' : '',
          error => ok = 'failed'
        ),
        finalize(() => {
          console.log(`${req.method} "${req.urlWithParams}" ${ok} in ${Date.now() - started} ms.`);
        })
      );
  }
}
```


Protección ante XSRF

- Cross-Site Request Forgery (XSRF) explota la confianza del servidor en la cookie de un usuario. HttpClient soporta el mecanismo “Cookie-to-Header Token” para prevenir ataques XSRF.
  - El servidor debe establecer un token en una cookie de sesión legible en JavaScript, llamada XSRF-TOKEN, en la carga de la página o en la primera solicitud GET. En las solicitudes posteriores, el cliente debe incluir el encabezado HTTP X-XSRF-TOKEN con el valor recibido en la cookie.
  - El servidor puede verificar que el valor en la cookie coincida con el del encabezado HTTP y, por lo tanto, asegúrese de que sólo el código que se ejecutó en su dominio pudo haber enviado la solicitud.
  - El token debe ser único para cada usuario y debe ser verificable por el servidor. Para mayor seguridad se puede incluir el token en un resumen de la cookie de autenticación de su sitio.
- Para establecer nombres de cookies / encabezados personalizados:

```
imports: [ // …
  HttpClientXsrfModule.withConfig({
        cookieName: 'My-Xsrf-Cookie',
        headerName: 'My-Xsrf-Header',
  }),]
```

- El servicio backend debe configurarse para establecer la cookie y verificar que el encabezado está presente en todas las solicitudes elegibles.

JWT: JSON Web Tokens

https://jwt.io

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier datos entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo si tenemos un servicio de envío de email otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

AuthService

```
@Injectable({providedIn: 'root'})
export class AuthService {
  private isAuth = false;
  private authToken: string = '';
  private name = '';
  constructor() {
    if (localStorage && localStorage.AuthService) {
      const rslt = JSON.parse(localStorage.AuthService);
      this.isAuth = rslt.isAuth;
      this.authToken = rslt.authToken;
      this.name = rslt.name;
    }
  }
  get AuthorizationHeader() { return this.authToken;  }
  get isAutenticated() { return this.isAuth; }
  get Name() { return this.name; }
  login(authToken: string, name: string ) {
    this.isAuth = true;
    this.authToken = authToken;
    this.name = name;
    if (localStorage) { localStorage.AuthService = JSON.stringify({isAuth, authToken, name}); }
  }
  logout() {
    this.isAuth = false;
    this.authToken = '';
    this.name = '';
    if (localStorage) { localStorage.removeItem('AuthService'); }
  }
}
```

LoginService

```
@Injectable({providedIn: 'root'})
export class LoginService {
  constructor(private http: HttpClient, private auth: AuthService) { }
  get isAutenticated() { return this.auth.isAutenticated;  }
  get Name() { return this.auth.Name;  }
  login(usr: string, pwd: string) {
    return new Observable(observable =>
      this.http.post('http://localhost:4321/login', { name: usr, password: pwd })
        .subscribe(
          data => {
            if(data['success']) { this.auth.login(data['token'], data['name']); }
            observable.next(this.auth.isAutenticated);
          },
          (err: HttpErrorResponse) => { observable.error(err); }
        )
    );
  }
  logout() { this.auth.logout(); }
}
```


ENRUTADO

Introducción

- El enrutado permite tener una aplicación de una sola página, pero que es capaz de representar URL distintas, simulando lo que sería una navegación a través páginas web, pero sin salirnos nunca de la página inicial. Esto permite:
  - **Memorizar rutas profundas dentro de nuestra aplicación**. Podemos contar con enlaces que nos lleven a partes internas (deeplinks), de modo que no estemos obligados a entrar en la aplicación a través de la pantalla inicial.
  - Eso **facilita también el uso natural del sistema de favoritos** (o marcadores) del navegador, así como el historial. Es decir, gracias a las rutas internas, seremos capaces de guardar en favoritos un estado determinado de la aplicación. A través del uso del historial del navegador, para ir hacia delante y atrás en las páginas, podremos navegar entre pantallas de la aplicación con los botones del navegador.
  - **Mantener y cargar módulos en archivos independientes**, lo que reduce la carga inicial y permite la carga perezosa.

Rutas internas

- En las URL, la “almohadilla“, el carácter “#“, sirve para hacer rutas a anclas internas: zonas de una página.
- Cuando se pide al navegador que acceda a una ruta creada con “#“ éste no va a recargar la página (cargando un nuevo documento que pierde el contexto actual), lo que hará es buscar el ancla que corresponda y mover el scroll de la página a ese lugar.
  - http://example.com/index.html
  - http://example.com/index. html#/seccion
  - http://example.com/index. html#/pagina_interna
- Es importante fijarse en el patrón "#/", sirve para hacer lo que se llaman "enlaces internos" dentro del mismo documento HTML.
- En el caso de Angular no habrá ningún movimiento de scroll, pues con Javascript se detectará el cambio de ruta en la barra de direcciones para intercambiar la vista que se está mostrando.
- Los navegadores HTML5 modernos admiten history.pushState, una técnica que cambia la ubicación y el historial de un navegador sin activar una solicitud de página del servidor. El enrutador puede componer una URL "natural" que es indistinguible de una que requeriría una carga de página.

Hash Bag

- Dado que los robots indexadores de contenido de los buscadores no siguen los enlaces al interior de la pagina (que asumen como ya escaneada), el uso del enrutado con # que carga dinámicamente el contenido impide el referenciado en los buscadores.
- Para indicarle al robot que debe solicitar el enlace interno se añade una ! después de la # quedando la URL:

```
http://www.example.com/index.html#!ruta
```


Angular Router

- El Angular Router ("router") toma prestado el modelo deeplinks. Puede interpretar una URL del navegador como una instrucción para navegar a una vista generada por el cliente y pasar parámetros opcionales en la ruta al componente para decidir qué contenido específico se quiere manejar.
- El Angular router es un servicio opcional que presenta la vista de un componente en particular asociado a una determinada URL.
- No es parte del núcleo Angular. Es un paquete de la biblioteca, @angular/router, a importar en el módulo principal como se haría con cualquier otro modulo Angular.

```
import { RouterModule, Routes } from '@angular/router';
```

- La aplicación tendrá un único router. Cuando la URL del navegador cambia, el router busca una correspondencia en la tabla de rutas para determinar el componente que debe mostrar.
- Las aplicaciones de enrutamiento deben agregar un elemento <base> index.html al principio de la etiqueta <head> para indicar al enrutador cómo componer las URL de navegación.

```
<base href="/">
```

Tabla de rutas

- La tabla de ruta es un conjunto de objetos Route.
- Toda ruta tiene una propiedad path con la ruta que se utiliza como patrón de coincidencia. Puede ser:
  - Única: *path: 'mi/ruta/particular'*
  - Parametrizada: *path: 'mi/ruta/:id'*
  - Vacía (solo una vez): *path: ''*
  - Todas las demás (solo una vez): *path: '**'*
- Dado que la búsqueda se realiza secuencialmente la tabla debe estar ordenada de rutas mas especificas a las mas generales.
- La ruta puede estar asociada a:
  - Un componente: *component: MyComponent*
  - Otra ruta (redirección): *redirectTo: '/otra/ruta'*
  - Otro módulo (carga perezosa): **loadChildren: 'ruta/otro/modulo'**
- Adicionalmente se puede indicar:
  - outlet: Destino de la ruta.
  - pathMatch: **prefix** | full
  - children: Subrutas
  - data: datos adicionales
  - Servicios canActivate, canActivateChild, canDeactivate, canLoad

Registrar enrutamiento

```
const routes: Routes = [
  { path: '', component: HomeComponent, pathMatch: 'full' },
  { path: 'path/:routeParam', component: MyComponent1 },
  { path: 'staticPath', component: MyComponent2 },
  { path: 'oldPath', redirectTo: '/newPath' },
  { path: 'path', component: MyComponent3 , data: { message: 'Custom' } },
  { path: '**', component: ErrorComponent },
];

@NgModule({
  imports: [ BrowserModule,
             RouterModule.forRoot(routes)
         ],
  // ...
})
export class AppModule { }
```

Directivas

- Punto de entrada por defecto:
  <**router-outlet**></router-outlet>
- Punto de entrada con nombre, propiedad outlet de la ruta:
  <router-outlet **name="aux"**></router-outlet>
- Generador de referencias href:
  <a **routerLink**="/path">
  <button **routerLink**="/">Go to home</button>
  <a [routerLink]="['/path', routeParam]">
  <a [routerLink]="['/path', { matrixParam: 'value' }]">
  <a [routerLink]="['/path']" [queryParams]="{ page: 1 }">
  <a [routerLink]="['/path']" fragment="anchor">
- Nombre de la Class CSS asociada a ruta actual:
  <a [routerLink]="['/path']" **routerLinkActive**="active"
  [routerLinkActiveOptions]="{exact: true}">

Trabajar con rutas

- Importar clases:
  import { Router, ActivatedRoute } from '@angular/router';
- Inyectar dependencias:
  constructor(private route: ActivatedRoute, private router: Router) { }
- Decodificar parámetros y ruta:
  - Valor actual (Instantánea):
    let id = this.route.snapshot.params['id'];
  - Detección de cambios (observable)
    route.url.map(segments => segments.join(''));
    let sid = this.route.queryParamMap.pipe(map(p => p.get('sid') || 'None'));
    this.token = this.route.fragment.pipe(map(f => f || 'None'));
- Navegación desde el código:
  this.router.navigate(['/ruta/nueva/1']);

Decodifica ruta

```
ngOnInit() {
  let id = this.route.snapshot.params['id'];
  if (id) {
    if (this.route.snapshot.url.slice(-1)[0].path === 'edit') {
      this.vm.edit(+id);
    } else {
      this.vm.view(+id);
    }
  } else if (this.route.snapshot.url.slice(-1)[0].path === 'add') {
    this.vm.add();
  } else {
    this.vm.load();
  }
 }
```


Parámetros observables

```
private obs$: any;
ngOnInit() {
  this.obs$ = this.route.paramMap.subscribe(
    (params: ParamMap) => {
    const id = +params.get('id'); // (+) converts string 'id' to a number
    if (id) {
      this.vm.edit(id);
    } else {
      this.router.navigate(['/404.html']);
    }
   });
 }
 ngOnDestroy() { this.obs$.unsubscribe(); }
```

Eventos de ruta

Evento	Desencadenado
NavigationStart	cuando comienza la navegación.
RoutesRecognized	cuando el enrutador analiza la URL y las rutas son reconocidas.
RouteConfigLoadStart	antes de que empiece la carga perezosa.
RouteConfigLoadEnd	después de que una ruta se haya cargado de forma perezosa.
NavigationEnd	cuando la navegación termina exitosamente.
NavigationCancel	cuando se cancela la navegación: un guardián devuelve falso.
NavigationError	cuando la navegación falla debido a un error inesperado.

- Durante cada navegación, el Router emite eventos de navegación a través de la propiedad observable Router.events a la se le puede agregar suscriptores para tomar decisiones basadas en la secuencia de eventos.

```
this.router.events.subscribe(ev => {
    if(ev instanceof NavigationEnd) { this.inicio = (ev as NavigationEnd).url == '/'; }
  });
```


Guardianes de rutas

- Los guardianes controlan el acceso a las rutas por parte del usuario o si se le permite abandonarla:
  - CanActivate: verifica el acceso a la ruta.
  - CanActivateChild: verifica el acceso a las ruta hijas.
  - CanDeactivate: verifica el abandono de la ruta, permitiendo descartar acciones pendientes que se perderán.
  - CanLoad: verifica el acceso a un módulo que requiere carga perezosa.
- Para crear un guardián hay que crear un servicio que implemente el correspondiente interfaz y registrarlo en cada ruta a controlar.

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
          boolean { return this.authService.isAutenticated;  }
}
{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard], },
```

Obtener datos antes de navegar

- En algunos casos es interesante resolver el acceso a datos antes de navegar para poder redirigir en caso de no estar disponibles.
- Servicio:

```
@Injectable({ providedIn: 'root', } )
export class DatosResolve implements Resolve<any> {
  constructor(private dao: DatosDAOService, private router: Router) {}
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<any> {
    return this.dao.get(+route.paramMap.get('id')).pipe(
      take(1),
      map(data => {
        if (data) { return data; } else { // id not found
          this.router.navigate(['/404.html']);
          return null;
        }
      }),
      catchError(err => { this.router.navigate(['/404.html']); return empty(); })
    );
  }
}
```


Obtener datos antes de navegar

- En la tabla de rutas:

```
{ path: 'datos/:id', component: DatosViewComponent,
    resolve: {elemento: DatosResolve} },
```

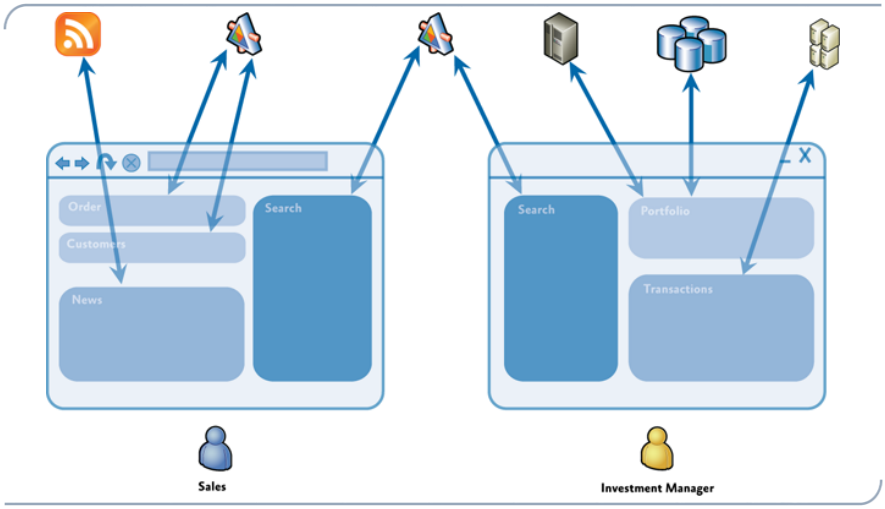
- En el componente:

```
ngOnInit() {
  this.route.data.subscribe((data: { elemento: any }) => {
    let e = data.elemento;
    // …
  });
```

Mantenimiento de estado

- Uno de los posibles aspectos problemáticos del enrutado es que los componentes se instancian con cada vista donde se estén usando, cada vez que se carga la ruta.
- Por este motivo todos los datos que se inicializan y se vuelven a poner a sus valores predeterminados cuando carga cualquier la vista.
- El mantenimiento de estado consiste en guardar los datos desde que el componente se destruye hasta que se vuelve a crear y se restaura el estado, situación antes de destruirse.
- A través de los servicios se puede implementar el mantenimiento de estado de los componentes en caso de ser necesario.
  - El patrón singleton, utilizado en la inyección de dependencias, asegura que sólo existe una instancia de ellos en el modulo, por lo que no pierden su estado, y, si hay varios componentes que dependen de un mismo servicio, todos recibirán la misma instancia del objeto.

Patrón Composite View



outlet

- Punto de entrada con nombre, propiedad outlet de la ruta:
  <router-outlet name="aside"></router-outlet>
- En la tabla de rutas:
  { path: 'mypath', component: MyComponent, outlet: 'aside' }
- Generador de referencias secundarias:
  <a [routerLink]="[{ outlets: { aside: ['mypath'] } }]">...
- Ruta múltiple:
  http://example.com/mainpath(aside:mypath)
- Navegación desde el código:
  this.router.navigate([{ outlets: { aside: ['mypath'] } }]);
- Eliminar el contenido:
  this.router.navigate([{ outlets: { aside: null } }]);

Lazy loading

- La carga diferida de módulos (lazy loading) se controla con el router:

```
export const routes: Routes = [
  // …
  { path: 'admin',
    loadChildren: './admin/admin.module#AdminModule' },
];
```

- El localizador loadChildren de carga perezosa es una cadena, no un tipo.
- La cadena identifica tanto el módulo de archivo (módulo TypeScript) como el módulo de clase (módulo Angular), este último separado del anterior por una #.
  ./admin/admin.module#AdminModule
  equivale a una importación en app.module
  import { AdminModule } from './admin/admin.module';

Lazy loading

- El módulo LazyLoad no debe estar entre los imports del @NgModule del AppModule.
- WebPack crea un bundle por cada módulo enrutado como loadChildren identificado por un número y la sub extensión .chunk
  - 0.chunk.js, 1.chunk.js, 2.chunk.js, …
- El módulo LazyLoad debe tener una tabla de rutas con la ruta vacía que indica el componente inicial del módulo.

```
export const routes: Routes = [
   { path: '', component: AdminMainComponent },
   { path: 'users', component: UsersComponent },
   { path: 'roles', component: RolesComponent },
];
```

- La tabla de rutas, en el módulo LazyLoad se debe importar a través del forChild:

```
@NgModule({  imports: [ RouterModule.forChild(routes), // …
```


DESPLIEGUE

Pruebas

- Son imprescindibles en entornos de calidad: permite ejecutar las pruebas unitarias, las pruebas de extremo a extremo o comprobar la sintaxis.
- Para comprobar la sintaxis: Hay que ejecutar el analizador con el comando:
  - ng lint
- Para ejecutar tests unitarios: Se puede lanzar los tests unitarios con karma con el comando:
  - ng test
- Para ejecutar tests e2e: Se puede lanzar los tests end to end con protractor con el comando:
  - ng e2e

Polyfill

- Angular se basa en los últimos estándares de la plataforma web. Dirigirse a una gama tan amplia de navegadores es un reto porque no todos son compatibles con todas las funciones de los navegadores modernos.
- Un Polyfill puede ser un segmento de código o un plugin que permite tener las nuevas funcionalidades u objetos de HTML5 y ES2015 en aquellos navegadores que nativamente no lo soportan.
- Es necesario activar (des-comentar) en el fichero polyfills.ts las funcionalidades utilizadas.

```
/** IE9, IE10 and IE11 requires all of the following polyfills. **/
import 'core-js/es6/symbol';
import 'core-js/es6/object';
// …
import 'core-js/es6/set';
// …
```

- Este archivo incorpora los polyfill obligatorios y muchos opcionales. Algunos polyfill opcionales se tendrá que instalar con npm.

Compilación

- Una aplicación Angular consiste principalmente en componentes y sus plantillas HTML. Debido a que los componentes y las plantillas proporcionadas por Angular no pueden ser entendidas directamente por el navegador, las aplicaciones de Angular requieren un proceso de compilación antes de que puedan ejecutarse en el navegador.
- Angular ofrece dos formas de compilar la aplicación:
  - Just-in-Time (JIT), que compila la aplicación en el navegador en tiempo de ejecución (la predeterminada cuando ejecuta los comandos de CLI ng build o ng serve).
  - Ahead-of-Time (AOT), que compila la aplicación antes de que la descargue el navegador.
- El compilador Ahead-of-Time (AOT) convierte el código TypeScript y el HTML de la aplicación Angular en un eficiente código JavaScript, durante la fase de despliegue antes de que el navegador descargue y ejecute la aplicación. Las ventajas de la compilación AOT son:
  - Representación más rápida: Con AOT, el navegador descarga una versión precompilada de la aplicación. El navegador carga directamente el código ejecutable para que pueda procesar la aplicación de inmediato, sin esperar a compilarla primero.
  - Menos peticiones asíncronas: El compilador incluye las plantillas HTML externas y las hojas de estilo CSS dentro de la aplicación JavaScript, eliminando solicitudes ajax separadas para esos archivos de origen.
  - Tamaño de descarga del framework de Angular más pequeño: No es necesario descargar el compilador Angular si la aplicación ya está compilada. El compilador ocupa aproximadamente la mitad de Angular, por lo que omitirlo reduce drásticamente la carga útil de la aplicación.
  - Detectar antes errores de plantillas: El compilador AOT detecta e informa de los errores de enlace de la plantilla durante el proceso de compilación antes de que los usuarios puedan verlos.
  - Mejor seguridad: AOT compila plantillas y componentes HTML en archivos JavaScript mucho antes de que se sirvan al cliente. Sin plantillas para leer y sin el riesgo de evaluación de HTML o JavaScript del lado del cliente, hay menos oportunidades para ataques de inyección.

Despliegue

- El despliegue más simple posible
  1. Generar la construcción de producción.
  2. Copiar todo dentro de la carpeta de salida (/dist por defecto) a una carpeta en el servidor.
  3. Configurar el servidor para redirigir las solicitudes de archivos faltantes a index.html.
- Construye la aplicación en la carpeta /dist
  - ng build
  - ng build --dev
- Paso a producción, construye optimizándolo todo para producción
  - ng build --prod
  - ng build --prod --env=prod
  - ng build --target=production --environment=prod
- Precompila la aplicación
  - ng build --prod –aot
- Cualquier servidor es candidato para desplegar una aplicación Angular. No se necesita un motor del lado del servidor para componer dinámicamente páginas de aplicaciones porque Angular lo hace en el lado del cliente.
- La aplicaciones Angular Universal y algunas funcionalidades especiales requieren una configuración especial del servidor.

https://angular.io/resources

UTILIDADES

Módulos de terceros

- Data Libraries
  - Angular Fire: The official library for Firebase and Angular
  - Apollo: Apollo is a data stack for modern apps, built with GraphQL.
  - Meteor: Use Angular and Meteor to build full-stack JavaScript apps for Mobile and Desktop.
  - ngrx: Reactive Extensions for angular2
- UI Components
  - ag-Grid: A datagrid for Angular with enterprise style features such as sorting, filtering, custom rendering, editing, grouping, aggregation and pivoting.
  - Angular Material 2: Material Design components for Angular
  - Clarity Design System: UX guidelines, HTML/CSS framework, and Angular components working together to craft exceptional experiences
  - Kendo UI: One of the first major UI frameworks to support Angular
  - ng-bootstrap: The Angular version of the Angular UI Bootstrap library. This library is being built from scratch in Typescript using the Bootstrap 4 CSS framework.
  - ng-lightning: Native Angular components & directives for Lightning Design System
  - ng2-bootstrap: Native Angular2 directives for Bootstrap
  - Onsen UI: UI components for hybrid mobile apps with bindings for both Angular 1 & 2.
  - Prime Faces: PrimeNG is a collection of rich UI components for Angular
  - Semantic UI: UI components for Angular using Semantic UI
  - Vaadin: Material design inspired UI components for building great web apps. For mobile and desktop.
  - Wijmo: High-performance UI controls with the most complete Angular support available.