



Angular 16

<https://angular.io/>

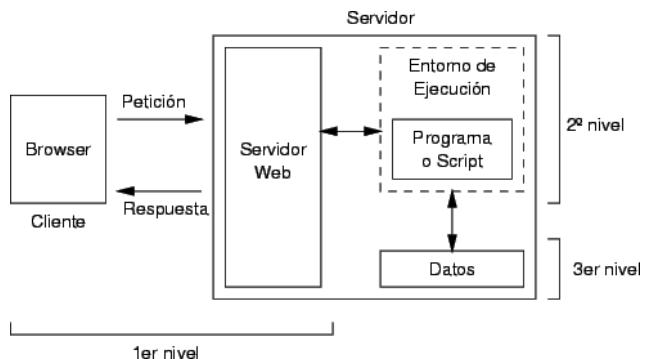
© JMA 2020. All rights reserved

INTRODUCCIÓN A ANGULAR 2+

© JMA 2020. All rights reserved

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).



© JMA 2020. All rights reserved

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 las 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).

© JMA 2020. All rights reserved

Introducción

- Angular (comúnmente llamado "Angular 2+" o "Angular 2", evolución de AngularJS) es un framework estructural para aplicaciones web de una sola página (SPA) siguiendo el patrón MVVM y orientación a componentes.
- Angular es una plataforma de desarrollo, construida sobre TypeScript. Como plataforma, Angular incluye:
 - Un marco basado en componentes para crear aplicaciones web escalables.
 - Una colección de bibliotecas bien integradas que cubren una amplia variedad de características, que incluyen enrutamiento, administración de formularios, comunicación cliente-servidor y más.
 - Un conjunto de herramientas para desarrolladores que ayudan a desarrollar, compilar, probar y actualizar el código.
- Angular es un framework que define 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 quieras 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.

© JMA 2020. All rights reserved

Características

- En AngularJS la escalabilidad estaba limitada por: el lenguaje JavaScript y la precarga del código.
- Uso de TypeScript: aporta orientación a objetos, validación, intellisense y refactoring.
- Uso de Anotaciones (metadatos): declarativa frente a imperativa.
- Desarrollo basado en 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 sencilla.
- El enlazado de datos y la inyección de dependencias eliminan gran parte del código que de otra forma se tendría que escribir.
- Incluye compilación AoT (Ahead of Time - Antes de tiempo) y también lazy-loading (descarga bajo demanda, una manera de sólo descargar los recursos realmente necesarios en cada momento).
- Está cada vez más orientado a grandes desarrollos empresariales.
- Es open source, apadrinado por Google.

© JMA 2020. All rights reserved

Características

- **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.
- **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.
- **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.
- **Pruebas:** Con Karma para pruebas unitarias, puede saber si ha roto cosas cada vez que guarda.
- **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.

© JMA 2020. All rights reserved

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.
- **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. Desde la aparición de Angular Universal (herramienta de renderizado de Angular desde el servidor), puede ejecutar Angular en el servidor.
- **Nativo:** Cree aplicaciones móviles nativas con estrategias de Cordova, Ionic o NativeScript. Permite crear aplicaciones híbridas, una solución low cost para aplicaciones móviles.
- **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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 Candidates (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

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Novedades 8.0

- Soporte SVG para las plantillas de los componentes.
- Carga diferencial por defecto: el navegador elige entre JavaScript moderno (es2015) o heredado (es5) en función de sus propias capacidades.
- Nueva sintaxis para la carga perezosa de rutas: Importaciones Dinámicas.
- Angular CLI Builders: Creación de comandos personalizados para Angular CLI.
- Soporte para Web Workers (subprocesos en segundo plan).
- Se ha eliminado todo el paquete @angular/http, hay que utilizar @angular/common/http en su lugar.
- Actualización de TypeScript 3.4.3
- Angular ha sido actualizado para usar v.6.4 de RxJS.

© JMA 2020. All rights reserved

Novedades 9.0

- Ivy: el nuevo compilador y motor de renderización activado por defecto:
 - Comprobación de tipos, errores de compilación y tiempos de compilación mejorados, que habilitan AOT de forma predeterminada
 - Tamaños de paquete más pequeños
 - Mejor depuración
 - Enlace de clase y estilo CSS mejorado
 - Internacionalización mejorada
 - Pruebas más rápidas
- Nuevas opciones para 'provideIn' del inyector de dependencias
- Nuevos componentes para incluir capacidades de YouTube y Google Maps
- Actualización a TypeScript 3.7
- Angular ha sido actualizado para usar la v6.5 de RxJS.

© JMA 2020. All rights reserved

Novedades 10.0

- Modo Estricto: este modo se puede poner cuando se crea el proyecto mediante el comando: `ng new --strict`
 - Usa el modo estricto del Typescript
 - Esto “impide” el uso de `any` para la definición de variables, mediante el linter
 - Las plantillas también se ponen en modo estricto
 - Se reducen los budgets de rendimiento en un 75%
 - Impide ciertos tipos de efectos colaterales al habilitar un tree shaking más avanzado.
 - Sólo recomendado para aquellos que ya tienen un nivel suficientemente alto en el manejo de Typescript y de los API's que se usen
- Se han reducido los navegadores soportados por defecto (se eliminan IE 9 y 10), para evitar cargar código compilado en es5.
- Actualización a TypeScript 3.9, TSLib 2.0 y TSLint 6.0, con un nuevo fichero de configuración `tsconfig.base.json`
- Se han eliminado más de 700 issues y se han tocado más de 2000.
- Nuevo Range Datepicker en Angular Material.

© JMA 2020. All rights reserved

Novedades 11.0

- Se han reducido los navegadores soportados por defecto eliminándose IE 11 y se eliminan definitivamente los IE 9, 10 y Mobile, para evitar cargar código compilado en es5.
- Mejoras del API de pruebas de componentes y mejoras de rendimiento
- Mejores informes de compilación (salida del CLI)
- Soporte actualizado de reemplazo de módulo en caliente en `ng serve`
- Angular Language Service actualizado, basado en Ivy
- Actualización a TypeScript 4.

© JMA 2020. All rights reserved

Novedades 12.0

- Desactivación del soporte para IE11 (se eliminará definitivamente en Angular v13)
- Agregadas las directivas de validación min y max a los formularios basados en plantillas.
- Soporte del operador de fusión nula (??) en las plantillas.
- Anuncio de que Protractor finaliza su desarrollo y queda obsoleto a finales del 2022 (junto con Angular v15).
- Transición desde los ID de mensajes de i18n heredados
- Mejoras en estilo, soporte de Sass, SCSS y LESS en las definiciones en línea de la propiedad styles de @Component
- Mejoras en la documentación
- El modo estricto está habilitado de forma predeterminada en la CLI.
- ng build ahora tiene como valor predeterminado producción.
- Soporte de Webpack 5 para producción.
- Actualización a TypeScript 4.2.
- La extensión Angular DevTools amplía las Chrome DevTools agregando capacidades específicas de depuración y generación de perfiles de Angular.

© JMA 2020. All rights reserved

Novedades 13.0

- Final definitivo del soporte y compatibilidad para IE11.
- Mejoras en el CLI, ahora admite el uso de caché de compilación persistente de forma predeterminada.
- Mejoras en el motor de compilación Ivy.
- Mejoras varias en el enrutador.
- Algunas mejoras importantes en TestBed para que limpie el DOM después de cada prueba.
- Mejoras y cambios significativos en los componentes de Angular Material para un mejor soporte de la accesibilidad (a11y).
- Actualización a TypeScript 4.4
- Angular ha sido actualizado para usar la v7.4 de RxJS.

© JMA 2020. All rights reserved

Novedades 14.0

- Autocompletar comandos en Angular CLI
- Accesibilidad optimizada del título de la página
- Comprobación estricta de tipos en los formularios reactivos
- Nuevas primitivas CDK
- Diagnóstico de desarrollador extendido
- Compilaciones de aplicaciones ESM experimentales
- Angular Devtools ahora también está disponible en Firefox
- Componentes aislados (preliminar)
- Versión preliminar de Micro Front End
- Actualización a TypeScript 4.6
- Angular ha sido actualizado para usar la v7.5 de RxJS.

© JMA 2020. All rights reserved

Novedades 15.0

- Standalone API con componentes, directivas y pipes
- Standalone API con el enrutador y HttpClient
- Mejoras en el tree-shakable (sacudir el árbol: eliminar código no utilizado)
- API de composición de directivas
- Optimización de imágenes con la directiva NgOptimizedImage
- Guardianes de rutas funcionales (como expresiones lambda)
- La carga Lazy desenvuelve las importaciones predeterminadas
- Mejoras en las trazas en DevTools
- Ahora se puede configurar las opciones predeterminadas para DatePipe
- Actualización a TypeScript 4.8

© JMA 2020. All rights reserved

Novedades 16.0

- Entradas obligatorias del componente: @Input({required: true})
- Autocompletar importaciones y etiquetas de cierre automático en plantillas
- Soporte para Trusted Types nativos que ayuda a prevenir ataques de cross-site scripting (XSS)
- ngOnDestroy flexible
- Soporte para la vinculación de datos del enrutador como entradas de componentes
- Compatibilidad con CSP para estilos en línea, Soporte para Tailwind CSS y Compatibilidad con el aislamiento de CSS.
- API de depuración de inyección de dependencias
- Herramientas mejoradas para Componentes, Directivas y Pipes Independientes
- Vista previa
 - Angular Signals mejora la reactividad y la detección de cambios
 - Nueva hidratación no destructiva de la aplicación completa y nuevas funciones de representación del lado del servidor
 - Sistema de compilación basado en esbuild
 - Soporte experimental para Jest y Web Test Runner
- Actualización a TypeScript 5.0

© JMA 2020. All rights reserved

Angular Platform



© JMA 2020. All rights reserved

Actualización

- El equipo de Angular ha 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.firebaseio.com/>

© JMA 2020. All rights reserved



TypeScript



<https://www.typescriptlang.org/>

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.).
- TypeScript amplia la sintaxis del JavaScript con la definición y comprobación de:
 - Tipos básicos: boolean, number, string, Any y Void.
 - Clases e interfaces
 - Herencia
 - Genéricos
 - Módulos
 - Anotaciones

© JMA 2020. All rights reserved

TypeScript

- El TypeScript se traduce (“transpila”, translate+compiler, compilar un lenguaje de alto nivel a otro de alto nivel o de niveles similares) 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: para cambiar de versión basta con configurar y volver a transpilar, dejando 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 puede instalar con el siguiente comando:
 - npm install -g typescript
- Podemos transpilar manualmente escribiendo
 - tsc helloworld.ts
- Los diversos “plug-in” se pueden descargar en: <https://www.typescriptlang.org>

© JMA 2020. All rights reserved

Comandos tsc

- Compilación continua:
 - tsc -w *.ts
 - tsc --watch *.ts
- Compilación de múltiples ficheros de entrada en un 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

© JMA 2020. All rights reserved

tsconfig.json

- El archivo tsconfig.json especifica los archivos raíz y las opciones del compilador necesarias para compilar el proyecto. La presencia de un archivo tsconfig.json en un directorio indica que el directorio es la raíz de un proyecto de TypeScript.

```
{  
  "compilerOptions": {  
    "target": "ES2015",  
    "module": "commonjs",  
    "strict": true,  
    "esModuleInterop": true,  
    "skipLibCheck": true,  
    "forceConsistentCasingInFileNames": true,  
    "paths": {  
      "@my/module": ["src/my/module/index.ts"]  
    }  
  },  
  "$schema": "https://json.schemastore.org/tsconfig",  
  "display": "Recommended"  
}
```

© JMA 2020. All rights reserved

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

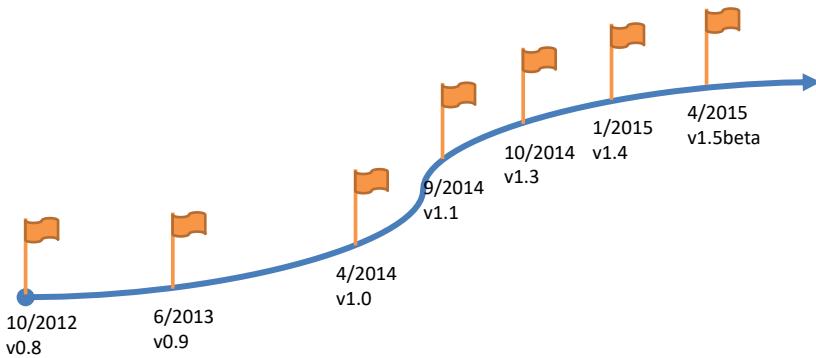
© JMA 2020. All rights reserved

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

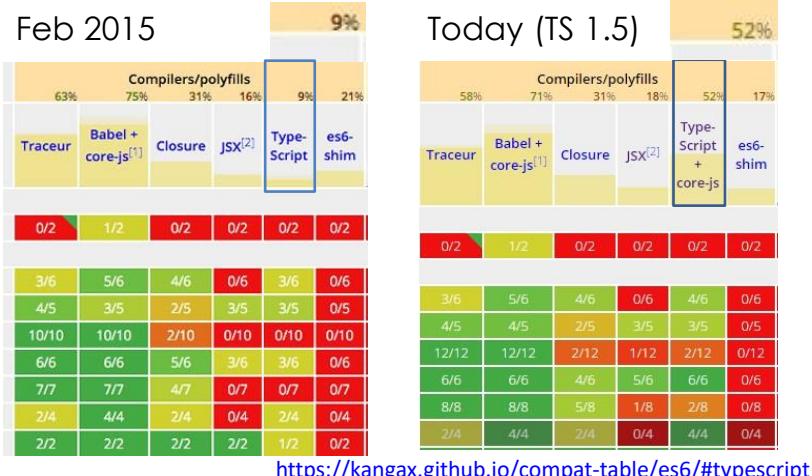
© JMA 2020. All rights reserved

Historia



© JMA 2020. All rights reserved

Compatibilidad con ES6



© JMA 2020. All rights reserved

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];
- Object: no es number, string, boolean, symbol, null, o undefined.
- Any, Unknown: var notSure: any = 4;
notSure = 'maybe a string instead';
notSure = false; // okay, definitely a boolean
var list:any[] = [1, true, 'free'];
- Void, Never: function warnUser(msg: string): void { alert(msg); }
- Null, Undefined
- Funciones, Clases e Interfaces

© JMA 2020. All rights reserved

Tipos especiales

- El tipo **any** representa cualquier valor, desactiva la verificación de tipos en aquellos casos una dependencia externa impide establecer tipos concretos.
- El tipo **unknown** representa cualquier valor. Es similar al tipo any, pero es más seguro porque no es legal hacer nada con un valor unknown.
- El tipo **void** es un pseudo tipo que representa la no devolución de valor por parte de una función o método. Solo se puede utilizar como tipo de retorno.
- El tipo **never** representa valores que nunca se observan. En un tipo de retorno de funciones, esto significa que la función lanza una excepción o termina la ejecución del programa.

© JMA 2020. All rights reserved

Definiciones

- Variables globales (no deberían usarse para evitar el acoplamiento):
`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'];`

© JMA 2020. All rights reserved

Control de tipos

- Las anotaciones de tipo en TypeScript son formas ligeras de registrar el contrato previsto de la variable o función, permitiendo en la compilación la validación de su correcto uso.

```
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;
```

© JMA 2020. All rights reserved

Sintaxis ampliada (ES6)

- Cadenas (Interpolación y múltiples líneas)

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

- Bucle foreach:

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

- Destructuring

```
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}
```

© JMA 2020. All rights reserved

Funciones con tipos

- 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 f(optional?: any, valor = 'foo') {  
    if(optional) {  
        :  
    }  
    :  
};
```

- Resto de los parámetros: Convierte una lista de parámetros en un array (Any por defecto).

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

© JMA 2020. All rights reserved

Funciones con tipos

- Se puede utilizar la desestructuración de parámetros para descomprimir convenientemente los objetos proporcionados como argumento en una o más variables locales en el cuerpo de la función.

```
function sum({ a, b, c }) {  
    console.log(a + b + c);  
}  
let obj = { a: 10, b: 3, c: 9 }  
sum(obj);
```

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

```
let str = 'foo';  
let chars = [ ...str ]; // [ 'f', 'o', 'o' ]  
let punto = { x: 10, y: 5 } // fn(...punto) equivale a fn(punto.x, punto.y)
```

© JMA 2020. All rights reserved

Sobrecargas de funciones

- Algunas funciones de JavaScript se pueden llamar con una variedad en número y tipos de argumentos. En TypeScript, podemos especificar que una función se puede llamar de diferentes formas definiéndole varias firmas. Para hacer esto, se escriben *firmas de sobrecarga* (dos o mas), precediendo a la *firma de implementación* con el cuerpo de la función (que no es visible desde el exterior y no se puede llamar directamente). La firma de implementación también debe ser compatible con las firmas de sobrecarga, el primer parámetro debe ser compatible con todas las sobrecargas (tipo unión o any) y se debe contemplar la variabilidad de los parámetros:

```
function makeDate(timestamp: number): Date; // firma de sobrecarga
function makeDate(m: number, d: number, y: number): Date; // firma de sobrecarga
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date { // firma de implementación
    if (d !== undefined && y !== undefined) {
        return new Date(y, mOrTimestamp, d);
    } else {
        return new Date(mOrTimestamp);
    }
}

const d1 = makeDate(12345678);
const d2 = makeDate(5, 5, 5);
const d3 = makeDate(1, 3); // firma de implementación
```

© JMA 2020. All rights reserved

Expresiones Lambda (Arrow functions)

- Las expresiones Lambda o arrow functions (en algunos lenguajes) son funciones anónimas.
- El operador => dispone de diferente firmas para la definición abreviada de las funciones anónimas:
 - Parámetro único: item => ...
 - Sin parámetros: () => ...
 - Varios parámetro: (a, b) => ...
 - Cuerpo como expresión (return implícito): (a, b) => a + b
 - Cuerpo con bloque de instrucciones (return explícito): () => { ...; return rsit; }

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

© JMA 2020. All rights reserved

Enumeraciones

- Las enumeraciones permiten a un desarrollador definir un conjunto de constantes con nombre. El uso de enumeraciones puede facilitar la documentación de la intención o restringir el conjunto de casos distintos.
- Las enumeraciones son una de las pocas características que tiene TypeScript que no es una extensión a nivel de tipo de JavaScript, desaparecen como tales en tiempo ejecución.
- TypeScript proporciona enumeraciones tanto numéricas (por defecto, base 0 con autoincremento para los miembros sin valor explícito) como basadas en cadenas.

```
enum Direction { Up = 1, Down, Left, Right, }

let direction: Direction = Direction.Left;
if(direction !== Direction.Up) {
    console.log(direction)
}

enum LogLevel {
    ERROR = 'error',
    WARN = 'warn',
    INFO = 'info',
    DEBUG = 'debug',
}
```

© JMA 2020. All rights reserved

Tipos Unión

- El sistema de tipos de TypeScript permite crear nuevos tipos combinando los existentes.
- Un tipo de unión es un tipo formado por dos o más tipos concatenados con `|` y que representan valores que pueden ser cualquiera de esos tipos.
- Nos referimos a cada uno de estos tipos como miembros de la unión.
- Los miembros de la unión pueden ser valores concretos como cadenas, objetos, ...
- TypeScript solo permitirá una operación si es válida para todos los miembros de la unión.
- La solución es estrechar la unión, que ocurre cuando TypeScript puede deducir un tipo más específico para un valor basado en la comprobación de tipos: `typeof`, `Array.isArray`, ...

```
type ModoCRUD = 'list' | 'add' | 'edit' | 'view' | 'delete';
let modo: ModoCRUD = 'list';

function f(x: number | number[], factor: number = 1) {
    if(typeof x === "number") {
        return x * factor;
    } else {
        let sum = 0;
        x.forEach(item => {
            sum += item;
        })
        return sum * factor;
    }
}

console.log(f(10, 2))
console.log(f([10, 20, 30]))
```

© JMA 2020. All rights reserved

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 (error del billón de dólares: null pointer exception).
 - Si se desea utilizar en un tipo el valor nulo o indefinido, se puede definir un tipo unión: tipo | null | undefined.

© JMA 2020. All rights reserved

Prevención de errores

- Operador para accesos de propiedad opcionales (resolución temprana de nulos)
`let x = foo?.bar.baz();`
- Operador de aserción no nulo (omite la comprobación de posible nulo)
`let x = cad!.toUpperCase();`
- Operador de fusión nula (valor por defecto para los nulos)
`let x = foo ?? bar();`
- En las veces que hay un valore sobre el que TypeScript no puede conocer su tipo, el operador as puede resolverlo.
`const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;`
- Funciones de aserción
`function assert(condition: any, msg?: string): asserts condition {
 if (!condition) { throw new AssertionError(msg) }
}
assert(typeof x === "number");`

© JMA 2020. All rights reserved

Objetos JavaScript

```
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
```

© JMA 2020. All rights reserved

Clases

- TypeScript ofrece un soporte completo para las clases de orientación a objetos.
- Permite implementar:
 - Atributos, métodos y propiedades
 - Encapsulación con los modificadores de acceso:
 - **public** ←*por defecto*
 - **private**
 - **protected** (*a partir de 1.3*)
 - Atributos de solo lectura con el modificador **readonly** (deben inicializarse en su declaración o en el constructor)
 - Sobre escritura de métodos con el modificador **Override**
 - Simula la sobrecarga de métodos a través de múltiples firmas (sobrecarga de funciones)
 - Miembros de clase, con el modificador **static** (name, length y call no son validos)
 - Herencia, implementación de interfaces, genéricos
 - Clases y miembros abstractos con el modificador **abstract**

© JMA 2020. All rights reserved

Clases

```
class Persona {  
    readonly MAYORIA_EDAD = 18;  
    private id: number;  
    private nombre: string;  
    private apellidos: string;  
    protected edad: number;  
    constructor(id: number, nombre: string, apellidos: string, edad: number) {  
        this.id = id;  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad;  
    }  
    public getId(): number { return this.id }  
    public getNombreCompleto(): string { return `${this.nombre} ${this.apellidos}` }  
    public esMayorDeEdad(): boolean { return this.edad >= this.MAYORIA_EDAD }  
    public cumpleAños(): void { if(!this.esMayorDeEdad()) this.edad++ }  
}  
  
let boy = new Persona(1, 'Peter', 'Pan', 10)  
let cadaver = Persona(2, 'Cruela', 'de Evil', 101) // Error de compilación
```

© JMA 2020. All rights reserved

Clases

- El acceso a los miembros dentro de la clase debe realizarse siempre a través de `this`.
- TypeScript ofrece una sintaxis especial para convertir un parámetro del constructor en un atributo de la clase con el mismo nombre y valor. Se crean anteponiendo al parámetro uno de los modificadores de visibilidad: `public`, `private`, `protected` o `readonly`. El atributo resultante se define con dicho modificador:
`constructor(private id: number, private nombre: string, private apellidos: string, protected edad: number) {}`
- Un inicializador puede dirigir la creación de la clase e inicializar los atributos de clase, tal y como pasa con los métodos de clase (`static`) no puede acceder a los miembros de instancia:

```
private static list: Array<Persona>;  
static {  
    this.list = []  
    this.list.push(new Persona(1, 'Peter', 'Pan', 10))  
    this.list.push(new Persona(2, 'Cruela', 'de Evil', 101))  
}
```

© JMA 2020. All rights reserved

Propiedades

- Las propiedades aportan toda una serie de ventajas:
 - Externamente se comportan como un atributo público.
 - Internamente están compuestas por dos métodos, con los descriptores de acceso get (función) y set (procedimiento), que controlan como entra y sale la información.
- TypeScript tiene algunas reglas especiales para los accesores:
 - Getters y setters deben tener el mismo tipo y la misma visibilidad de miembro.
 - Si no se especifica el tipo del parámetro setter, se deduce del tipo de retorno del getter.
 - Las propiedades de solo lectura solo disponen del método get.
 - No tienen que tener correspondencia directa con los atributos.

```
private nombre: string;
public get Nombre(): string {
    return this.nombre.toUpperCase();
}
public set Nombre(valor: string) {
    if (this.nombre === valor) return;
    if (!valor)
        throw new RangeError('null value');
    this.nombre = valor;
    this.onPropertyChanged('Nombre');
}
if(boy.Nombre === "") {
    boy.Nombre = 'Peter'
}
```

© JMA 2020. All rights reserved

Compila a ES5

```
var Persona = /** @class */ (function () {
    function Persona(id, nombre, apellidos, edad) {
        this.MAYORIA_EDAD = 18;
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
    Object.defineProperty(Persona.prototype, "Id", {
        get: function () { return this.id; },
        enumerable: false,
        configurable: true
    });
    Persona.prototype.getNombreCompleto = function () { return "".concat(this.nombre, " ").concat(this.apellidos); };
    Persona.prototype.esMayorDeEdad = function () { return this.edad >= this.MAYORIA_EDAD; };
    Persona.prototype.cumpleAños = function () { if (!this.esMayorDeEdad()) this.edad++; };
    return Persona;
}());
```

© JMA 2020. All rights reserved

También a ES6 o superior

```
class Persona {  
    constructor(id, nombre, apellidos, edad) {  
        this.MAYORIA_EDAD = 18;  
        this.id = id;  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad;  
    }  
    get Id() { return this.id; }  
    getNombreCompleto() { return `${this.nombre} ${this.apellidos}`; }  
    esMayorDeEdad() { return this.edad >= this.MAYORIA_EDAD; }  
    cumpleAños() { if (!this.esMayorDeEdad()) this.edad++; }  
}
```

© JMA 2020. All rights reserved

Herencia

- Las clases pueden extender una clase base (solo una). Una clase derivada tiene todos los miembros de su clase base y puede definir miembros adicionales o sobre escribir métodos heredados.

```
class Adulito extends Persona {  
    readonly EDAD_JUBILACION;  
    constructor(id: number, nombre: string, apellidos: string, edad: number) {  
        super(id, nombre, apellidos, edad)  
        this.EDAD_JUBILACION = 67;  
    }  
    public sePuedeJubilar(): boolean { return this.edad >= this.EDAD_JUBILACION }  
    public override cumpleAños(): void { if (this.esMayorDeEdad()) this.edad++; else super.cumpleAños(); }  
}
```

- Si la clase base tiene un constructor, el heredero deberá llamar super(); en el cuerpo de su constructor antes de usar cualquier miembro this. Acepta la sobre escritura de métodos pero deben ir marcados con Override si está activada directiva -nolmplicitOverride. La clase base puede ser abstracta (no instanciable), en cuyo caso puede tener métodos abstractos:

```
abstract class Persona {  
    abstract getTitulo(): string;
```

© JMA 2020. All rights reserved

Genéricos

- Las clases e interfaces genéricas tienen una lista de parámetros de tipo genérico entre corchetes angulares (<>) después del nombre.

```
class Elemento<T> {  
    constructor(private key: T, private value: string) {}  
    public get Key() { return this.key; }  
    public get Value() { return this.value; }  
}  
  
let genero = new Elemento<string>('M', 'Masculino')  
let provincia = new Elemento(28, 'Madrid')  
let tipo: string = genero.Key  
let id: number = provincia.Key;
```

- Se puede inferir los tipos de los parámetros de tipo si aparecen en el constructor.
- Se puede usar una restricción para limitar los tipos que puede aceptar un parámetro de tipo.

```
class Elemento<T extends number | string> {
```

© JMA 2020. All rights reserved

Funciones genéricas

- Las funciones genéricas se implementan declarando parámetros de tipo en la firma de la función:

```
function firstElement<Type>(arr: Type[]): Type | undefined { return arr[0]; }  
const s = firstElement(["a", "b", "c"]); // s is of type 'string'  
const n = firstElement([1, 2, 3]); // n is of type 'number'  
const u = firstElement([]); // u is of type undefined
```

- El tipo es inferido, elegido automáticamente, por TypeScript. Se puede inferir de los parámetros de entrada y de tipo de retorno.

```
function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {  
    return arr.map(func);  
}  
const parsed = map(["1", "2", "3"], (n) => parseInt(n)); // Input is 'string' and Output is 'number[]'
```

- Se puede usar una restricción para limitar los tipos que puede aceptar un parámetro de tipo.

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
```

© JMA 2020. All rights reserved

Interfaces

- Al igual que otros lenguajes orientados a objetos, TypeScript permite la definición e implementación de interfaces pero con matices:

```
interface Pingable {  
    ping(): void;  
}  
class Sonar implements Pingable {  
    ping() {  
        console.log("ping!");  
    }  
}
```
- Los tipos interface desaparecen en tiempo de ejecución (no se pueden comprobar con instanceof o similares), son restricciones para la compilación. Las clases no están obligadas a implementarlos pero si a cumplirlos, es solo una verificación de que la clase se puede tratar como el tipo de interfaz.

© JMA 2020. All rights reserved

Interfaces: Propiedades solo lectura y opcionales

```
interface SquareConfig {  
    readonly id: number;  
    color?: string;  
    width?: number;  
}  
  
function createSquare(config: SquareConfig): { color: string; area: number } {  
    let newSquare = { color: 'white', area: 100 };  
    if (config.color) {  
        newSquare.color = config.color;  
    }  
    if (config.width) {  
        newSquare.area = config.width * config.width;  
    }  
    return newSquare;  
}  
  
let mySquare = createSquare({ id: 1, color: 'black' });
```

© JMA 2020. All rights reserved

Interfaces: Como prototipos

- Usando interfaces:

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

- Definición en línea, sin interfaces:

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

© JMA 2020. All rights reserved

Interfaces: Funciones y Arrays

- Prototipos o firmas de Funciones

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

- Prototipos de Arrays o colecciones

```
interface StringArray {  
    [index: number]: string;  
}  
interface StringDictionary {  
    [index: string]: string; // Para poder usar la sintaxis "indexado" (obj["key"]) además de la de "punto" (obj.key)  
}  
let myArray: StringArray = ['Bob', 'Fred'];  
let myDictionary: StringDictionary = { fila: '5', columna: 'C'}
```

© JMA 2020. All rights reserved

Extensión Interfaces y Tipos de intersección

- Herencia de interfaces

```
interface Colorful {  
    color: string;  
}  
interface Circle {  
    radius: number;  
}  
interface ColorfulCircle extends Colorful, Circle {  
    active: boolean;  
}  
const cc: ColorfulCircle = { color: "red", radius: 42, active: true };
```

- Tipos de intersección, similares a los interfaces pero con sutiles diferencias

```
type ColorfulCircle = Colorful & Circle;  
const cc: ColorfulCircle = { color: "blue", radius: 42 }
```

© JMA 2020. All rights reserved

Módulos

- A partir de ECMAScript 2015, JavaScript tiene un concepto de módulos. TypeScript contaba con su propio concepto de módulos y espacios de nombres (módulos internos) que adecuó para adoptar el propuesto por ECMAScript. También da soporte al cargador de Node.js para módulos CommonJS y el cargador RequireJS para módulos AMD en aplicaciones web.

```
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");
```

© JMA 2020. All rights reserved

Módulos (ES6)

- Los módulos se basan en el concepto de ficheros como módulos:
 - Cualquier archivo que contenga un nivel superior import o export se considera un módulo.
 - Solo se puede importar lo previamente exportado, lo no exportado es privado del módulo
 - Es necesario importar antes de utilizar
 - El fichero en el from sin extensión y ruta relativa (./ ../) o sin ruta (NODE_MODULES)
- Para exportar:

```
export public class MyClass {}  
export default class MyDefaultClass {}  
export { MY_CONST, myFunction, name as otherName }
```
- Para importar:

```
import * from './my_module';  
import * as MyModule from './my_module';  
import MyDefaultClass, { MyClass, MY_CONST, myFunction as func } from './my_module';
```
- Directorios como módulos (Agrupación, Reexportaciones, Barrel): Importar y exportar (index.ts)

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

© JMA 2020. All rights reserved

Decoradores (ES7)

- Existen ciertos escenarios que requieren funciones adicionales para admitir la anotación o modificación de clases y miembros de clase. Los decoradores proporcionan una forma de agregar anotaciones y una sintaxis de metaprogramación para declaraciones de clase y miembros.
- Un decorador es un tipo especial de declaración que se puede adjuntar a una declaración de clase, método, descriptor de acceso, atributo o parámetro. Los decoradores usan el formato @expression, donde expresión se debe evaluar como una función que se llamará en tiempo de ejecución con los metadatos sobre la declaración decorada.

```
function anotacion(value: string) {  
    // configura la función decorador devuelta  
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {  
        // este es el decorador, hace algo con 'target' (elemento anotado) y 'value'...  
    };  
}  
@anotacion('valor')  
abstract class Persona {  
    @notBlank  
    private nombre: string;
```

© JMA 2020. All rights reserved

async/await (ES2017)

- La sintaxis `async/await` permite un estilo de programación secuencial en procesos asíncronos, delegando en el compilador o interprete su implementación.
- La declaración de función `async` define una función asíncrona, que devuelve un objeto `AsyncFunction`. Una función asíncrona es una función que opera asíncrónicamente a través del bucle de eventos, utilizando una promesa implícita para devolver su resultado. Pero la sintaxis y la estructura de su código usando funciones asíncronas se parece mucho más a las funciones síncronas estándar.
- El operador `await` se usa para esperar a una `Promise` y sólo dentro de una `async function`.

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => { setTimeout(() => { resolve(x); }, 2000); });
}

async function f1() {
  let x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}
```

© JMA 2020. All rights reserved

Compatibilidad con Frameworks JS

- Typescript permite comunicarse con código JavaScript ya creado e incluso añadirle “tipos” a través de ficheros de definiciones `.d.ts` que prototipan los tipos que reciben y devuelven las funciones de una librería.
- Ya existen miles de ficheros de definiciones (`.d.ts`) que incluyen la práctica totalidad de los frameworks más populares (<http://definitelytyped.org/>).
 - `npm install --save-dev @types/jquery`
- Definiciones: <https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types>

```
1 declare module Sayings {
2   class Greeter {
3     greeting: string;
4     constructor(message: string);
5     greet(): string;
6   }
7 }
8 declare var greeter: Sayings.Greeter;
9 
```

© JMA 2020. All rights reserved

Angular esta escrito en TypeScript

```
import { Component, OnInit } from '@angular/core';
import { LoggerService } from 'src/lib/my-core';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
  title = 'Hola mundo';

  constructor(private log: LoggerService) { }

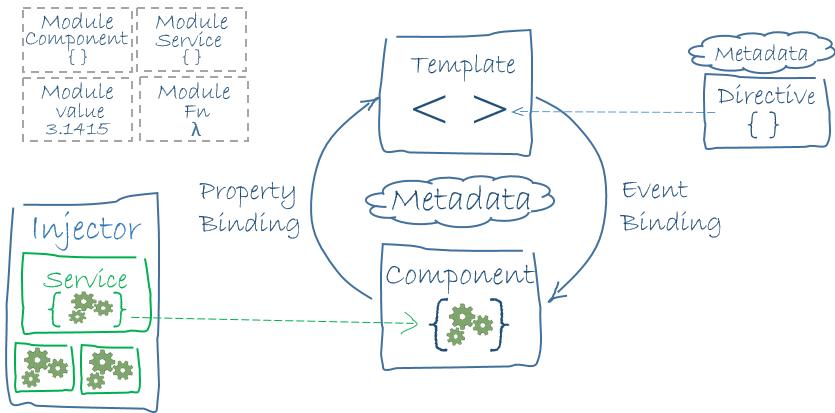
  ngOnInit(): void {
  }
}
```

© JMA 2020. All rights reserved

ARQUITECTURA DE ANGULAR

© JMA 2020. All rights reserved

Arquitectura de Angular



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 { }
```

© JMA 2020. All rights reserved

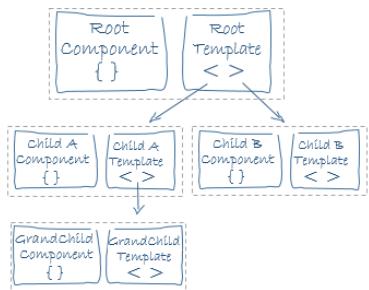
Componentes

- Un componente controla una parte de la pantalla denominada vista.
- Se puede considerar que un componente 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.

© JMA 2020. All rights reserved

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.

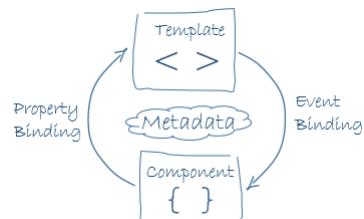
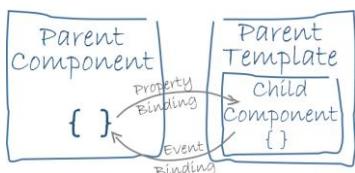


Una plantilla puede utilizar otros componentes, actúan como contenedores, dichos componentes pueden contener a su vez otros componentes. Esto crea un árbol de componentes: el modelo de composición.

© JMA 2020. All rights reserved

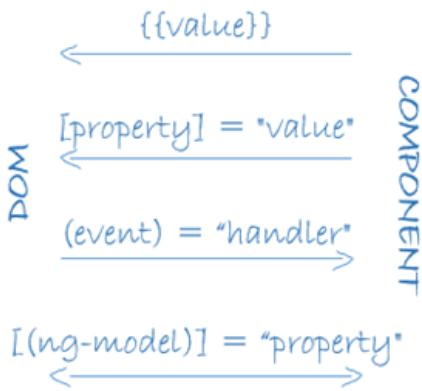
Enlace de datos

- 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.



© JMA 2020. All rights reserved

Enlace de datos



- La comunicación entre Componente y Plantilla puede ser:
 - Unidireccional:
 - Datos ($C \rightarrow P$)
 - Interpolación
 - Propiedades
 - Comandos ($P \rightarrow C$)
 - Eventos
 - Bidireccional:
 - Datos ($P \leftrightarrow C$)
 - Directivas

© JMA 2020. All rights reserved

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 debe manipular el árbol DOM es en las directivas", para que entre dentro del ciclo de vida de compilación, binding y renderización del HTML que sigue Angular.
- Aunque un componente es técnicamente una directiva-con-plantilla, conceptualmente son dos elementos completamente diferentes.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 específica 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).

© JMA 2020. All rights reserved

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 injector dispone de un contenedor con las instancias de servicios que crea él mismo. Si una instancia no está en el contenedor, el injector crea una nueva y la añade al contenedor antes de devolver el servicio a Angular, por eso los servicios son singlettons en el ámbito de su injector.
- 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.

© JMA 2020. All rights reserved

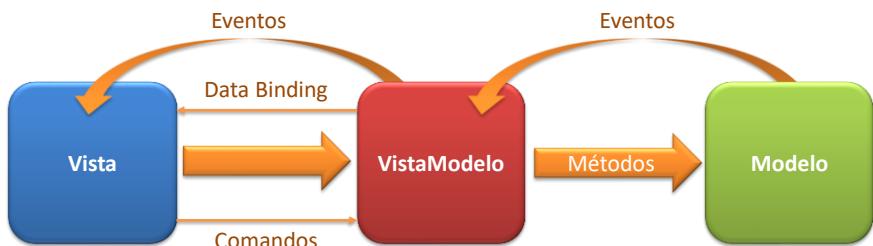
Angular Elements

- 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 instantiable 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.

© JMA 2020. All rights reserved

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 **VistaViewModel** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



© JMA 2020. All rights reserved

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)

© JMA 2020. All rights reserved

HERRAMIENTAS DE DESARROLLO

© JMA 2020. All rights reserved

IDEs

- Visual Studio Code - <http://code.visualstudio.com/>
 - VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.
 - Visual Studio Code for the Web provides a free, zero-install Microsoft Visual Studio Code experience running entirely in your browser: <https://vscode.dev>
- 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

© JMA 2020. All rights reserved

Extensiones Visual Studio Code

- [Spanish Language Pack for Visual Studio Code](#)
- [Angular Essentials](#)
- [Auto Close Tag](#)
- [Auto Rename Tag](#)
- [Code Spell Checker](#)
 - [Spanish - Code Spell Checker](#)
- [IntelliSense for CSS class names](#)
- [Path Intellisense](#)

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

Angular DevTools

- Angular DevTools es una extensión de Chrome y Firefox que proporciona capacidades de depuración y creación de perfiles para aplicaciones Angular. Angular DevTools es compatible con Angular v12 y superior, con Ivy habilitado. Está disponible en la [Chrome Web Store](#) y en [Firefox Addons](#).
- La pestaña Componentes permite explorar la estructura de la aplicación. Se puede visualizar e inspeccionar las instancias de directivas y componentes, así como obtener una vista previa o modificar su estado.
- La pestaña Profiler permite obtener una vista previa de la ejecución de la detección de cambios de Angular y comprender cuál es el cuello de botella en el rendimiento durante la ejecución de la detección de cambios.

© JMA 2020. All rights reserved

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.
 - `npm install -g @angular/cli`

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

Schematics en Angular CLI (v.6)

- Schematics es una herramienta de flujo de trabajo para la web moderna; permite aplicar transformaciones al proyecto, 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`

© JMA 2020. All rights reserved

Nuevos comandos (v.6)

- **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.
- **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`

© JMA 2020. All rights reserved

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 el comando:
 - ng e2e

© JMA 2020. All rights reserved

Despliegue

- Construye la aplicación en la carpeta /dist
 - ng build --dev
- Paso a producción, construye optimizándolo todo para producción
 - ng build
 - ng build --env=prod
 - ng build --target=production --environment=prod
- Compila "antes de tiempo" la aplicación
 - ng build --aot
- Para ejecutar el generador de despliegue asociado al proyecto:
 - ng add @angular/fire
 - ng deploy

© JMA 2020. All rights reserved

Estructura de directorios de soporte

```
src  
e2e  
.editorconfig  
.gitignore  
angular.json  
package-lock.json  
package.json  
README.md  
tsconfig.json  
tsconfig.app.json  
tsconfig.spec.json
```

- src: Fuentes de la aplicación
- ~~e2e: Test end to end (v12)~~
- Ficheros de configuración de librerías y herramientas
- Posteriormente aparecerán los siguientes directorios:
 - .angular: Cache de las compilaciones (ng cache)
 - node_modules: Librerías y herramientas descargadas
 - dist: Resultado para publicar en el servidor web (ng build)
 - coverage: Informes de cobertura de código (ng test)
- tsconfig.xxxx.json: Configuración del compilador TypeScript para la aplicación y las pruebas.

© JMA 2020. All rights reserved

Estructura de directorios de aplicación

```
app  
  app-routing.module.ts  
  app.component.css  
  app.component.html  
  app.component.spec.ts  
  app.component.ts  
  app.module.ts  
assets  
environments  
  environment.prod.ts  
  environment.ts  
favicon.ico  
index.html  
main.ts  
styles.css
```

- app: Carpeta que contiene los ficheros fuente principales de la aplicación.
- assets: Carpeta con los recursos que se copiarán a la carpeta build.
- environments: ~~configuración de los diferentes entornos. (v15)~~
- main.ts: Arranque del módulo principal de la aplicación. No es necesario modificarlo.
- favicon.ico: Ícono para el navegador.
- index.html: Página principal (SPA).
- style.css: Se editarán 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.~~

© JMA 2020. All rights reserved

Configurar valores específicos del entorno

- Crear el directorio src/environments/
- Crear los ficheros environment.ts, environment.prod.ts, ...

```
export const environment = {
  production: false
};
```
- Importar siempre el fichero sin sub extensión:

```
import { environment } from '../environments/environment';
```
- Configurar reemplazos de archivos específicos de destino en angular.json:

```
"configurations": {
  "production": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.prod.ts"
      }
    ],
  }
},
```

© JMA 2020. All rights reserved

ESLint

- ESLint (<https://eslint.org/>) es una herramienta para identificar e informar sobre patrones encontrados en código ECMAScript/JavaScript, con el objetivo de hacer que el código sea más consistente y evitar errores.
- Se puede instalar ESLint usando npm:
 - `npm install eslint --save-dev`
- Luego se debe crear un archivo de configuración `.eslintrc.json` en el directorio, se puede crear con `--init`:
 - `npx eslint --init`
- Se puede ejecutar ESLint con cualquier archivo o directorio:
 - `npx eslint **/*.js`
- Para habilitarlo y ejecutarlo en las últimas versiones de Angular:
 - `ng add @angular-eslint/schematics`
 - `ng lint`

© JMA 2020. All rights reserved

e2e: Cypress

- Escribir pruebas e2e requiere muchas herramientas diferentes trabajando juntas. Cypress es un todo en uno, no es necesario instalar 10 herramientas y bibliotecas independientes para configurar el entorno de pruebas. Han tomado algunas de las mejores herramientas de su clase y las han hecho funcionar juntas sin problemas.
- Las pruebas de Cypress solo están escritas en JavaScript, el código de prueba se ejecuta dentro del propio navegador.
- Para habilitarlo y ejecutarlo en las últimas versiones de Angular:
 - ng add @cypress/schematic
 - ng e2e
 - ng g spec
- Para evitar conflictos entre Jasmine (unit tests) y Chai (e2e tests), añadir al principio del tsconfig.json de la raíz del proyecto:

```
{ "exclude": ["cypress", "./cypress.config.ts"],
```
- Y en el tsconfig.json del directorio cypress, sustituir "include": ["**/*.ts"], por:

```
"include": ["**/*.ts", "cypress", "./cypress.config.ts"],  
"exclude": [],
```

© JMA 2020. All rights reserved

Jest

- Hay un schema disponible para:
 - instalar Jest, tipos y un builder
 - agregar archivos de configuración de Jest
 - modificar la configuración en package.json, angular.json y tsconfig.spec.json
 - eliminar Karma y Jasmine junto con sus archivos de configuración
- Para sustituir Karma y Jasmine junto por Jest:
 - ng add @briebug/jest-schematic
- Las pruebas se siguen ejecutando con:
 - ng test

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Librerías de terceros

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

```
"scripts": [  
  "./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": "/" }  
,
```

© JMA 2020. All rights reserved

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>`

© JMA 2020. All rights reserved

Espacios de trabajo, proyectos, aplicaciones, librerías

- Creación de una aplicación (proyecto único)
 - `ng new nombre-aplicación`
- Creación de un espacio de trabajo (multi proyectos):
 - `ng new my-workspace --create-application=false`
 - Añadir un proyecto de tipo aplicación:
 - `cd my-workspace`
 - `ng generate application my-first-app`
 - `ng serve my-first-app`
 - Añadir un proyecto de tipo librería:
 - `ng generate library my-lib`
 - `ng build my-lib --prod`
 - `cd dist/my-lib`
 - `npm publish`

© JMA 2020. All rights reserved

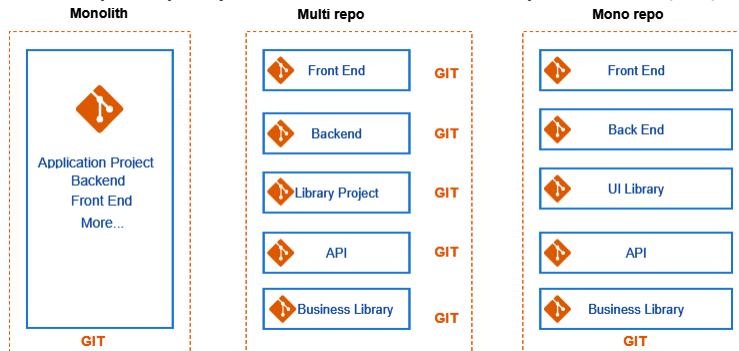
Espacios de trabajo, proyectos y librerías

- Un espacio de trabajo contiene los archivos de uno o más proyectos. Un proyecto es el conjunto de archivos que comprenden una aplicación independiente o una biblioteca que se puede compartir.
- Por defecto genera para desarrollar en "repositorios múltiples" en el que cada aplicación reside en su propio espacio de trabajo. Angular CLI también admite espacios de trabajo con múltiples proyectos para desarrollos "monorepo".
- Para configurar un espacio de trabajo de varios proyectos:
 - ng new my-workspace --create-application false
 - cd my-workspace
 - ng generate application my-first-app
 - ng generate library my-lib
- Para usar compilaciones incrementales en caliente la librería en la aplicación:
 - ng build my-lib --watch

© JMA 2020. All rights reserved

Estilos de desarrollo

- Monolito: un proyecto en un repositorio.
- Múltirepo: múltiples proyectos, un repositorio por proyecto.
- Monorepo: múltiples proyectos en un único repositorio (nx)



© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

MÓDULOS

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

@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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 {
```

© JMA 2020. All rights reserved

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 los 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.
- Opcionalmente, configurar un nombre público en tsconfig.json
 - "paths": { "@my/core": ["src/lib/my-core/index.ts"] }

© JMA 2020. All rights reserved

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 (componentes de UI)

© JMA 2020. All rights reserved

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 inyector no lo encuentra, como debería ser, suministrará un null al constructor. El decorador `@Optional` permite que el constructor no reciba el parámetro.

© JMA 2020. All rights reserved

Componentes independientes (v15)

- Los componentes, directivas y pipes independientes proporcionan una forma simplificada de crear aplicaciones Angular al reducir la necesidad de NgModules, incluso prescindir de ellos.
- Los componentes, directivas y pipes independientes deben especificar explícitamente sus dependencias con la propiedad imports: componentes, directivas y pipes independientes. Es posible que algunas de las dependencias no se marquen como independientes, sino que se declaren y exporten mediante un NgModule, en cuyo caso también pueden importar NgModules con los artefactos dependientes.
- Los componentes, directivas y pipes de los módulos solo pueden ser importados o cargados diferidamente a través de los módulos que los exportan.
- Se pueden combinar módulos (con artefactos dependientes) con artefactos independientes y viceversa. Conceptualmente, los componentes, directivas y pipes independientes se pueden ver como módulos con un solo artefacto exportado. Los componentes, directivas y pipes independientes se pueden importar con la propiedad imports del @NgModule y no en la propiedad declarations reservada para los que pertenecen al módulo. Las aplicaciones existentes pueden adoptar de forma incremental y opcional el nuevo estilo independiente.

© JMA 2020. All rights reserved

Arranque de la aplicación

- Modular:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
```

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

- Operación de arranque independiente:

```
import {bootstrapApplication} from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent);
```

© JMA 2020. All rights reserved

SPA

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Curso de Angular</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>
```

© JMA 2020. All rights reserved

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 el enrutamiento, navegar con URLs ...
HttpClientModule	@angular/common/http	Para acceder a un servidor

© JMA 2020. All rights reserved

SERVICIOS

© JMA 2020. All rights reserved

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 específica para los servicios, no hay una clase base ni un lugar para registrarlos, son simples Clases.
- Tienen la siguiente particularidad:
 - Pueden ir anotadas con `@Injectable()`, que 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.
 - Si implementan `OnDestroy`, el método `ngOnDestroy` permite cualquier limpieza personalizada que deba realizarse cuando se destruya la instancia.

© JMA 2020. All rights reserved

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() { ... }
}
```

© JMA 2020. All rights reserved

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 singlettons en el ámbito de su inyector.
- El provider es cualquier cosa que pueda crear o devolver un servicio, como la propia clase que define el servicio. Los providers pueden registrarse en los módulos y en cualquier nivel del árbol de componentes de la aplicación a través de sus metadatos. Los providers de un módulo se importan cuando se importa el módulo.

© JMA 2020. All rights reserved

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],
  viewProviders: [EmojiService]
  ...
})
export class AppComponent {
```

© JMA 2020. All rights reserved

Autoreferenciado 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' })
```
- Con el valor 'platform', proporciona una instancia única compartida por todas las aplicaciones en la página.
- Permiten a las herramientas de optimización sacudir los árboles, lo que elimina los servicios que la aplicación no está usando.

© JMA 2020. All rights reserved

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' }]

© JMA 2020. All rights reserved

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] }]
```

© JMA 2020. All rights reserved

Inyectores funcionales

- La Inyección de Dependencias se realiza habitualmente a través de los parámetros del constructor (ej: notify). Con la función inject() del @angular/core se puede realizar la inyección en:
 - El cuerpo del constructor de una clase que está siendo instanciada por el sistema DI (ej: log).
 - En el inicializador para campos de dichas clases (ej: http).
 - En la función factory especificada para useClass de un Provider, @Injectable, InjectionToken o interceptor.

```
@Injectable({providedIn: 'root'})  
export class DatosService {  
  private http: HttpClient = inject(HttpClient);  
  private log: LoggerService;  
  constructor(private notify: NotificationService) {  
    this.log = inject(LoggerService, { optional: true });  
  }  
  :  
}
```

- Las llamadas a la función inject() fuera del contexto de creación de clases a través del inyector darán como resultado un error. La función inject() utiliza el inyector actualmente activo.

© JMA 2020. All rights reserved

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 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)
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Inyectores Explícitos

- Habitualmente no tenemos que crear los inyectores en Angular, dado que se crea automáticamente un injector a nivel del módulo para toda la aplicación durante el proceso de arranque.
- Los componentes cuentan con sus propios inyectores.
- Se puede crear un injector de forma explícita:
 - const injector = Injector.create([HttpClient, Logger]);
 - const injector = createEnvironmentInjector([HttpClient, Logger]); // v14
- Para posteriormente obtener instancias inyectadas:
 - let srv= injector.get(Logger);
- Las dependencias son únicas dentro del alcance de un injector, sin embargo, al ser Angular DI un sistema de inyección jerárquica, los inyectores anidados pueden crear sus propias instancias de servicio.

© JMA 2020. All rights reserved

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, y forman en paralelo un árbol de los inyectores.
- Cuando un componente solicita una dependencia, Angular intenta satisfacela 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 (o entorno del arranque independiente). Si se queda sin antepasados y no está registrado en el módulo o entorno, se genera un error.
- En los componentes, los servicios registrados con viewProviders de no son visibles para el contenido proyectado (contenido de la etiqueta componente mostrado con <ng-content>). El contenido proyectado comienza la búsqueda a partir de los providers del componente, ignorando los viewProviders.

© JMA 2020. All rights reserved

Modificar la jerárquica de inyectores

- Cuando Angular crea un componente independiente, crea un inyector independiente separado para garantizar que los proveedores importados por el componente estén "aislados" del resto de la aplicación, no pueden "filtrar" sus detalles de implementación al resto de la aplicación.
- Usando el decorador @Self, el inyector solo mira los proveedores del propio componente.

```
constructor(@Self() public localLogger: Logger){}
```
- El decorador @SkipSelf omite el inyector local y buscar en la jerarquía para encontrar un proveedor que satisfaga esta dependencia.

```
constructor(@SkipSelf() public generalLogger: Logger){}
```
- La anotación @Host() impide el burbujeo y obliga a que el componente anfitrión tenga registrado el proveedor.

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

© JMA 2020. All rights reserved

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<CoreModule> {  
    return {  
        ngModule: CoreModule,  
        providers: [{provide: MyServiceConfig, useValue: config }]  
    };  
}
```
- En el servicio:

```
constructor(@Optional() config: MyServiceConfig) {  
    if (config) { ... }  
}
```
- En el módulo principal:

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

© JMA 2020. All rights reserved

Arranque independiente

- El módulo principal dispone de la propiedad `providers` para registrar los servicios e importa los providers de los módulos importado.
- La operación de arranque independiente se basa en la configuración explícita de una lista de proveedores para la inyección de dependencias. La configuración se realiza a través de la propiedad `providers` del segundo parámetro de `bootstrapApplication()`.
- En Angular, las funciones de los módulos prefijadas con `provide` se usan para importar los providers de un módulos. Si un módulo no ofrece dichas funciones, se puede usar la utilidad `importProvidersFrom()` con el módulo para importar sus providers en contextos independientes.
- Para inicializar la aplicación (anteriormente en el constructor del módulo principal), se usa el nuevo token múltiple `ENVIRONMENT_INITIALIZER`.

```
bootstrapApplication(AppComponent, {  
    providers: [  
        { provide: ERROR_LEVEL, useValue: environment.ERROR_LEVEL },  
        { provide: ENVIRONMENT_INITIALIZER, multi: true, useValue: () => inject(AppService).init() }  
        provideRouter(MAIN_ROUTES), importProvidersFrom(SecurityModule)  
    ]  
});
```

© JMA 2020. All rights reserved

COMPONENTES

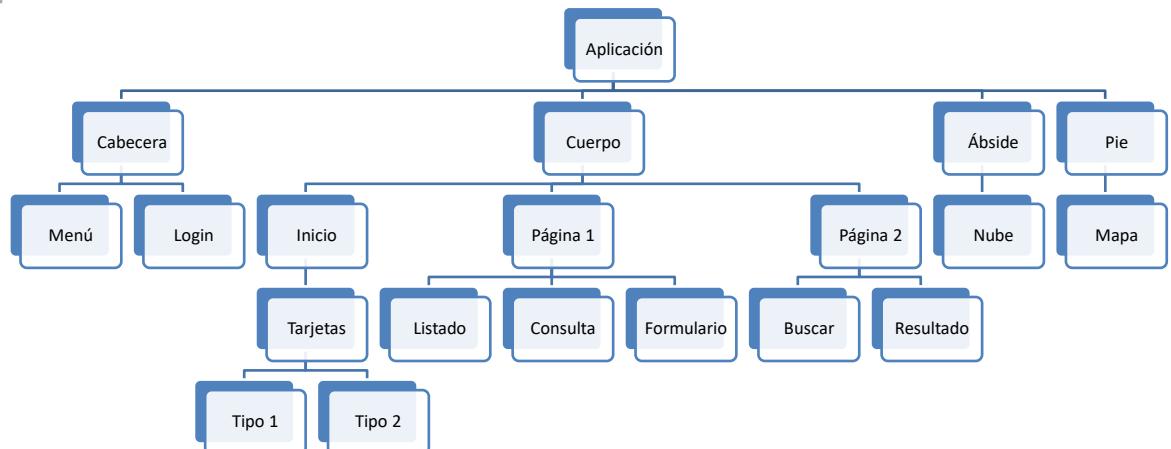
© JMA 2020. All rights reserved

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 son etiquetas, se usan en las plantillas y establecen un cauce bien definido de entrada/salida para su comunicación con otros componentes.

© JMA 2020. All rights reserved

Árbol de componentes



© JMA 2020. All rights reserved

Clasificación

- Componentes de aplicación:
 - Son componentes íntimamente ligados a la aplicación cuya existencia viene determinada por la existencia de la aplicación.
 - Estos componentes deben favorecer el desacoplamiento de entre la plantilla, el estilo y la clase del componente.
- Componentes compartidos (wedged, controles, ...):
 - Son componentes de bajo nivel, que existen para dar soporte a los componentes de aplicación pero son independientes de aplicaciones concretas.
 - Estos componentes requieren un mayor control sobre la plantilla y la estética por los que el acoplamiento será mayor, así como la dependencia entre ellos.
 - En muchos casos serán implementaciones de terceros que se incorporaran como módulos externos.

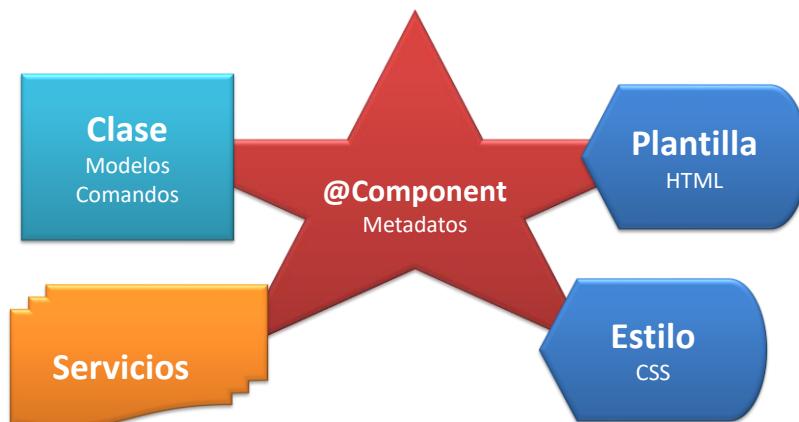
© JMA 2020. All rights reserved

Características

- Cada componente consta de:
 - Un selector de CSS que define la etiqueta el componente cuando se usa en una plantilla
 - Una clase de TypeScript que define el comportamiento.
 - Una plantilla HTML que declara lo que se representa en la página.
 - Opcionalmente, estilos CSS aplicados a la plantilla.
- Las clases de los componentes:
 - Están anotadas con `@Component`
 - Que exponen datos como atributos o propiedades (modelos) y funcionalidad como métodos (comandos) para su consumo por la plantilla
 - Opcionalmente, exponen propiedades (`@Input`) y eventos (`@Output`) para interactuar con otros componentes.

© JMA 2020. All rights reserved

Definición



© JMA 2020. All rights reserved

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';
  }
}
```

© JMA 2020. All rights reserved

@Component

- **selector:** Selector CSS (Tag HTML) con el prefijo app- ("prefix" en angular.json) para los de aplicación 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).
- **providers:** Lista de los proveedores disponibles para este componente y sus hijos.
- **viewProviders** - Lista de los proveedores disponibles para este componente y las vistas de sus hijos.

© JMA 2020. All rights reserved

Registrar

- Globalmente:

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

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

• En el contenedor (obsoleto v.9+):

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

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

© JMA 2020. All rights reserved

Componentes independientes (v15)

- Los componentes se pueden marcar como standalone: true y no necesitan declararse en un NgModule (da error si se intenta), pero deben especificar explícitamente sus dependencias con la propiedad imports (NgModules y componentes, directivas y pipes independientes).

```
@Component({
  selector: 'photo-gallery',
  standalone: true,
  imports: [ImageGridComponent, MatButtonModule, nglf, ngForOf],
  template: `
    <image-grid [images]="imageList"></image-grid>
    <button mat-button>Next Page</button>
  `,
})
export class PhotoGalleryComponent {
```

- Se han refactorizado las directivas y pipes integrados de los módulos estándar de Angular a sus versiones independientes para proporcionar una forma directa de importación.

© JMA 2020. All rights reserved

Propiedades enlazables

- Para suministrar los contenidos a la plantilla, la clase del componente puede exponer atributos públicos o protegidos y propiedades. Las propiedades aportan toda una serie de ventajas:
 - Externamente se comportan como un atributo público.
 - Internamente están compuestas 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 (_fullName === newName) return;  
        if (this.validate(newName)) {  
            this._fullName = newName;  
            this.fullNameChanged();  
        } else {  
            // ...  
        }  
    }  
    :  
}
```

© JMA 2020. All rights reserved

Comandos y funciones

- La clase del componente implementa su funcionalidad a través de métodos y expone algunos de ellos, como comandos o funciones interpolables, para su enlazado en la plantilla.
 - Los comandos son procedimientos (no devuelven valor) destinados a ser enlazados a eventos como controladores de eventos. Pueden contar con tantos parámetros como se consideren oportunos. Uno de ellos puede ser el objeto event del estándar DOM con toda la información de acompañamiento.
 - Las plantillas disponen de cierta capacidad de cálculo, las funciones interpolables realizan todos aquellos cálculos que exceden la capacidad de las plantillas o afectan a reglas de negocios. Puede recibir tantos argumentos como se consideren oportuno y obligatoriamente deben devolver el valor por el que les sustituirá la plantillas. No deberían realizar cambios, solo consultas.

© JMA 2020. All rights reserved

Objeto event (DOM)

Propiedad/ Método	Devuelve	Descripción
altKey	Boolean	Devuelve true si se ha pulsado la tecla ALT y false en otro caso
bubbles	Boolean	Indica si el evento pertenece al flujo de eventos de bubbling
button	Entero	El botón del ratón que ha sido pulsado.
cancelable	Boolean	Indica si el evento se puede cancelar
cancelBubble	Boolean	Indica si se ha detenido el flujo de eventos de tipo bubbling
charCode	Entero	El código unicode del carácter correspondiente a la tecla pulsada
clientX	Entero	Coordenada X de la posición del ratón respecto del área visible de la ventana
clientY	Entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana
ctrlKey	Boolean	Devuelve true si se ha pulsado la tecla CTRL y false en otro caso

© JMA 2020. All rights reserved

Objeto event (DOM)

Propiedad/ Método	Devuelve	Descripción
currentTarget	Element	El elemento que es el objetivo del evento
detail	Entero	El número de veces que se han pulsado los botones del ratón
eventPhase	Entero	La fase a la que pertenece el evento: 0 – Fase capturing 1 – En el elemento destino 2 – Fase bubbling
isChar	Boolean	Indica si la tecla pulsada corresponde a un carácter
keyCode	Entero	Indica el código numérico de la tecla pulsada
metaKey	Entero	Devuelve true si se ha pulsado la tecla META y false en otro caso
pageX	Entero	Coordenada X de la posición del ratón respecto de la página
pageY	Entero	Coordenada Y de la posición del ratón respecto de la página
preventDefault()	Función	Se emplea para cancelar la acción predefinida del evento

© JMA 2020. All rights reserved

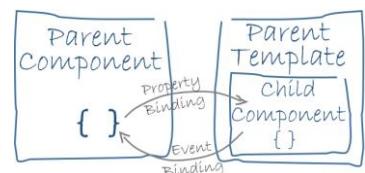
Objeto event (DOM)

Propiedad/Método	Devuelve	Descripción
relatedTarget	Element	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)
screenX	Entero	Coordenada X de la posición del ratón respecto de la pantalla completa
screenY	Entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
shiftKey	Boolean	Devuelve true si se ha pulsado la tecla SHIFT y false en otro caso
stopPropagation()	Función	Se emplea para detener el flujo de eventos de tipo bubbling
target	Element	El elemento que origina el evento
timeStamp	Número	La fecha y hora en la que se ha producido el evento
type	Cadena	El nombre del evento

© JMA 2020. All rights reserved

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: con {required: true} es obligatorio
`@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>();
<my-comp [(size)]="fontSize"></my-comp>`



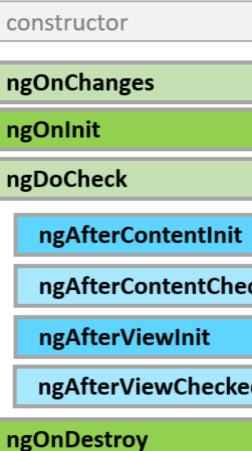
© JMA 2020. All rights reserved

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.fontSize.px]="size">FontSize: {{size}}px</label>
  </div>`})
export class SizerComponent {
  @Input() size: number | string = 12;
  @Output() sizeChange = new EventEmitter<number>();
  dec(): void { this.resize(-1); }
  inc(): void { this.resize(+1); }
  resize(delta: number): void {
    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>
```

© JMA 2020. All rights reserved

Ciclo de vida



- Cada componente tiene un ciclo de vida gestionado por el Angular.
- Angular lo crea y pinta, proyecta su contenido, 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.

© JMA 2020. All rights reserved

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 inicialice las vistas del componente y sus hijos (@ViewChild).
ngAfterViewChecked	Responder después de que Angular chequee las vistas del componente y sus hijos.
ngOnDestroy	Limpiar justo antes de que Angular destruya el componente.

© JMA 2020. All rights reserved

ESTILOS

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 prolífica 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.

© JMA 2020. All rights reserved

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;  
}
```

© JMA 2020. All rights reserved

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: `@import './variables.css';`).

© JMA 2020. All rights reserved

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.ShadowDom  
})
```

- Los modos de encapsulación disponibles son:

- **ShadowDom**: 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 proporciona ningún tipo de encapsulación de estilo CSS, todos los estilos proporcionados del componente son aplicables a cualquier elemento HTML de la aplicación.

© JMA 2020. All rights reserved

Precedencia (de mayor a menor)

- Cuando hay varios enlaces al mismo nombre de clase o propiedad de estilo, Angular usa un conjunto de reglas de precedencia para resolver conflictos y determinar qué clases o estilos se aplican finalmente al elemento. Cuanto más específico sea, mayor será su precedencia.
 1. Enlaces de plantillas
 1. Enlace de propiedad (por ejemplo, <div [class.foo]="'hasFoo'"> o <div [style.color]="'color'">)
 2. Enlace de mapa (por ejemplo, <div [class]="'classExpr'"> o <div [style]="'styleExpr'">)
 3. Valor estático (por ejemplo, <div class="foo"> o <div style="color: blue">)
 2. Enlaces de directivas anfitrionas
 1. Enlace de propiedad (por ejemplo, host: {'[class.foo]': 'hasFoo'} o host: {'[style.color]': 'color'})
 2. Enlace de mapa (por ejemplo, host: {'[class]': 'classExpr'} o host: {'[style]': 'styleExpr'})
 3. Valor estático (por ejemplo, host: {'class': 'foo'}) o host: {'style': 'color: blue'}
 3. Enlaces de componentes anfitriones
 1. Enlace de propiedad (por ejemplo, host: {'[class.foo]': 'hasFoo'} o host: {'[style.color]': 'color'})
 2. Enlace de mapa (por ejemplo, host: {'[class]': 'classExpr'} o host: {'[style]': 'styleExpr'})
 3. Valor estático (por ejemplo, host: {'class': 'foo'} o host: {'style': 'color: blue'})

© JMA 2020. All rights reserved

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.
(v12) La opción de compilación `inlineStyleLanguage` permite definir el lenguaje de la hoja de estilo en los estilos en línea. Las opciones de idioma admitidas actualmente son CSS (predeterminado), Sass, SCSS y LESS.

© JMA 2020. All rights reserved

Novedades

- Angular v11.2 agregó soporte nativo para ejecutar TailwindCSS PostCSS con Angular CLI.
- Angular v12 introdujo la opción de usar Sass en línea en los componentes.
- Angular v12 también introdujo Critical CSS Inlining para ayudar a garantizar que las aplicaciones Angular entreguen las mejores métricas posibles de Core Web Vital (Para Google, son un factor SEO más a evaluar para medir la experiencia de usuario).
- Angular v13 eliminó el soporte para IE11, lo que hace posible que Angular adopte un estilo web moderno como CSS Grid, variables CSS (propiedades personalizadas) de CSS, CSS calc(), ::hover y más.

© JMA 2020. All rights reserved

Configuración

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

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

© JMA 2020. All rights reserved

PLANTILLAS

© JMA 2020. All rights reserved

Plantillas

- En Angular, las plantillas se escriben en HTML (y SVG a partir de la versión 8) 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.
- A partir de la versión 16 se da soporte a las etiquetas con autocierre.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 más 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.

© JMA 2020. All rights reserved

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}}

© JMA 2020. All rights reserved

Operador de fusión nula ?? (v12).

- El operador de fusión nula (??) es una forma fluida y conveniente para suministrar un valor alternativo cuando el valor al que se aplica es nulo.
- En las plantillas, se puede usar la nueva sintaxis para simplificar condicionales complejos:

```
 {{age! == null && age! == undefined ?  
   age : calculateAge ()}}
```
- Se convierte en:

```
 {{ edad ?? calculateAge ()}}
```

© JMA 2020. All rights reserved

Operador de aserción no nula !.

- 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.

```
<p *ngIf="item">Color: {{item!.color}}</p>
```
- El operador de aserción no nula no protege contra nulo o indefinido, si title es null o undefined se produce un 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}}
```

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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}]`
`{{pi | number: '3.2-5'}}`
- **currency:** muestra un número con el símbolo de la moneda local y con el número de decimales correctos.
`{{ importe | currency }}` `{{ importe | currency:'EUR': 'symbol':'4.2-2' }}`
- **percent:** muestra un número como un porcentaje.
`{{ ratio | percent}}` `{{ ratio | percent:'2.2-2'}}`
- **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):
`{{miFecha | date:'dd/MM/yyyy'}}`
Formato por defecto: `{ provide: DATE_PIPE_DEFAULT_OPTIONS, useValue: { dateFormat: 'dd/MMM/yy' } },`

© JMA 2020. All rights reserved

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>
<li *ngFor="let n of list|slice:1:10">{{n}}`
- **keyvalue**: transforma un objetos en una colección de pares nombre-valor (key:value):
`<li *ngFor="let a of map | keyvalue">{{item.key}}: {{item.value}}`

© JMA 2020. All rights reserved

Pipes Predefinidos

- **i18nPlural**: transforma un valor numérico a una cadena que pluraliza el valor de acuerdo con un mapa (tipo valor:literal donde el valor se expresa como '=valor exacto' y 'other' para el resto) y las reglas de configuración regional.
`{{ messages.length | i18nPlural:{'=0': 'Sin mensajes.', '=1': 'Un mensaje.', '=2': 'Dos mensajes.', 'other': '# mensajes.' } }}`
- **i18nSelect**: selector genérico que muestra la cadena asociada que coincide con el valor actual.
`{{sexo | i18nSelect:{'M':'invitalo', 'F':'invitala', 'other':'invitale' } }}`
- **json**: convierte un objeto JavaScript en una cadena JSON, solo utilizado para depuración.

© JMA 2020. All rights reserved

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 || maxlen < 2 || !value || value.length <= maxlen) ? value :
      (value.substr(0, maxlen - 1) + '\u2026');
  }
}
<div>{{ descripcion | elipsis:100 }}</div>
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Optimizar con Pipes puros

- Una función en una expresión de plantilla se invoca cada vez que se ejecuta la detección de cambios.
- Debido a esto, se considera mejor sustituir la llamada a la función en la expresión con un pipe puro que encapsula su lógica.
- La mejora en el rendimiento proviene del hecho de que Angular optimiza los pipes puros llamando a transform solo cuando cambian el valor al que se aplica o los parámetros adicionales que de la función de transformación. Si alguno de estos valores cambia, se vuelve a ejecutar el método transform del pipe.
- Se puede crear un pipe genérico que invoque a las funciones:

```
@Pipe({ name: 'exec' })
export class ExecPipe implements PipeTransform {
  transform(fn: Function, ...args: any[]): any { return fn(...args); }
}
```
- Para sustituir:
 - {{ calcula(a, b) }}
- Por:
 - {{ calcula | exec:a:b }}

© JMA 2020. All rights reserved

AsyncPipe

- El AsyncPipe (impuro) acepta un Promise o un 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">
```

© JMA 2020. All rights reserved

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	<code>{{expression}}</code> <code>[target] = "expression"</code> bind-target = "expression"	Interpolación Propiedad Atributo Class, Style
Unidireccional de la plantilla a la fuente de datos	<code>(target) = "statement"</code> on-target = "statement"	Evento
En ambos sentidos	<code>[(target)] = "expression"</code> bindon-target = "expression"	Bidireccional

© JMA 2020. All rights reserved

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).
- El atributo HTML (tiempo de diseño) y la propiedad DOM (tiempo de ejecución) 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 (forma en la que está escrito el documento HTML).
- Los atributos pueden inicializar propiedades, siendo las propiedades las que luego pueden cambiar.
- 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).

© JMA 2020. All rights reserved

Objetivo de un enlace de datos

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

© JMA 2020. All rights reserved

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>`

© JMA 2020. All rights reserved

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 + ' | ';`
`}`

© JMA 2020. All rights reserved

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 las directivas sea el único sitio donde se debería 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.

© JMA 2020. All rights reserved

ngNonBindable

- Para mostrar llaves en una plantilla y evitar los errores de compilación, es necesario una secuencia de escape:

```
<p>  
  Esto evita que se evalue: {{"{{"}} 1 + 1 {"}}}"}}  
</p>
```

- La directiva ngNonBindable desactiva la interpolación, las directivas y los enlaces en las plantillas para el contenido (elementos secundarios) del elemento host pero no afecta a la interpolación, las directivas y los enlaces del propio elemento host.

```
<p [hidden]="" EstoSiSeEvalua" ngNonBindable>  
  Es no se evalúa: {{ 1 + 1 }}  
</p>
```

© JMA 2020. All rights reserved

ngPreserveWhitespace

- La eliminación de espacios en blanco (caracteres espacio, tabuladores, saltos de líneas, ..) puede reducir en gran medida el tamaño del código generado por AOT y acelerar la creación de vistas. A partir de Angular 6, el valor predeterminado para preserveWhitespaces es falso (se eliminan los espacios en blanco).
- De forma predeterminada, el compilador AOT elimina los espacios en blanco de la siguiente manera:
 - Recorta todos los espacios en blanco al principio y al final de una plantilla.
 - Elimina los nodos de texto con solo espacios en blanco.
 - Reemplaza una serie de caracteres de espacio en blanco en los nodos de texto con un solo espacio.
 - NO altera los nodos de texto dentro de las etiquetas HTML como `<pre>` o donde son significativos como en el contenido de `<textarea>`.
- La directiva ngPreserveWhitespace permite conservar los espacios en blanco en ciertos fragmentos de una plantilla.
`<div ngPreserveWhitespace> </div>`

© JMA 2020. All rights reserved

Estilos: ngStyle

- Mediante enlazado:

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

© JMA 2020. All rights reserved

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>
```

© JMA 2020. All rights reserved

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 público 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, // ...`

© JMA 2020. All rights reserved

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'.
```

© JMA 2020. All rights reserved

## NgOptimizedImage [ngSrc]

- La métrica Largest Contentful Paint (LCP) o Despliegue del contenido más extenso es una métrica importante centrada en el usuario para medir la velocidad de carga percibida porque marca cuando es probable que el contenido principal de la página se haya cargado. Un LCP rápido ayuda a asegurar al usuario que el la página sea útil.
  - NgOptimizedImage garantiza que la carga de imágenes se priorice (LCP) mediante: configuración automática del atributo fetchpriority en la etiqueta <img>, carga perezosa de imágenes no prioritarias por defecto y comprobar que hay una etiqueta <link rel="preload" ...> correspondiente en el encabezado del documento.
  - Además, genera la URL de activos apropiados si se proporciona una función ImageLoader, genera automáticamente un srcset. Requiere que width y height están configurados y advierte si se han configurado incorrectamente o si la imagen se distorsionará visualmente cuando se renderice.
  - La directiva NgOptimizedImage está marcada como independiente y se debe importar directamente (módulo o componente). Se aplica a <img> con ngSrc y dispone de propiedades adicionales como width, height, ngSrcset, sizes, loading y priority
- ```
<img ngSrc="graph.jpg" width="400" height="200" ngSrcset="100w, 200w" priority />
```

© JMA 2020. All rights reserved

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]="expresión">
    <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>
```

© JMA 2020. All rights reserved

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>
```

© JMA 2020. All rights reserved

ngPlural (v2.4)

- Diseñada para la pluralización, selecciona una plantilla en función a un valor numérico:
 - **ngPluralCase**: Expresión valor (=valor exacto, 'other' para el resto), si coinciden la etiqueta existirá.

```
<div [ngPlural]="messages.length">
  <ng-template ngPluralCase="=0">Sin mensajes.</ng-template>
  <ng-template ngPluralCase="=1">{{messages[0].nombre}}</ng-template>
  <ng-template ngPluralCase="few">algunos</ng-template>
  <ng-template ngPluralCase="other">
    <ul>
      <li *ngFor="let message of messages">
        {{message.nombre}}
      </li>
    </ul>
  </ng-template>
</div>
```

© JMA 2020. All rights reserved

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 colección".
- La palabra clave let permite crear 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>
```

© JMA 2020. All rights reserved

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; }
```

© JMA 2020. All rights reserved

Directiva iterativa: *ngForOf

- Para llenar 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>
```

© JMA 2020. All rights reserved

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 elementos como opciones:

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

© JMA 2020. All rights reserved

*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;
```

© JMA 2020. All rights reserved

ng-content

- La directiva <ng-content> es un elemento de transclusión (trasladar e incluir) del contenido de la etiqueta del componente a la plantilla (proyección).

```
<card [header]="">Este es el contenido a transcluir que puede contener otras etiquetas y  
componentes</card>
```

```
@Component({  
  selector: 'card',  
  template: `  
    <div class="card">  
      <H1>{{ header }}</H1>  
      <ng-content></ng-content>  
    </div>`,  
})  
export class CardComponent {  
  @Input() header: string = 'this is header';
```

© JMA 2020. All rights reserved

ng-content

- <ng-content> acepta un atributo select que permite definir un selector CSS que indica el contenido que se muestra en la directiva.
- Con el uso del atributo select se puedan definir varios ng-content, uno por cada tipo de contenido.
- Selector asociado a un atributo de HTML:
`<ng-content select="[titulo]"></ng-content>`
- Selector asociado a un class de CSS:
`<ng-content select=".cuerpo"></ng-content>`
- Contenido:
`<card [header]=""Hola mundo"">
 <div titulo <div class="cuerpo" componentes</div>
</card>`

© JMA 2020. All rights reserved

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">`

© JMA 2020. All rights reserved

SVG como plantillas

- Se pueden utilizar archivos SVG como plantillas de los componentes y usar directivas o enlaces al igual que con las plantillas HTML. Esto permite generar dinámicamente gráficos interactivos.

```
<svg width="100%" height="100vh">
  <g id="R1" transform="translate(250 250)">
    <ellipse rx="100" ry="0" opacity=".7" [attr.fill]="fillColor" (click)="changeColor()">
      <animate attributeName="cx" dur="8s" values="-20; 220; -20" repeatCount="indefinite" />
      <animate attributeName="ry" dur="3s" values="10; 60; 10" repeatCount="indefinite" />
      <animateTransform attributeName="transform" type="rotate" dur="7s" from="0" to="360" repeatCount="indefinite" />
    </ellipse>
  </g>
  <use xlink:href="#R1" transform="rotate(72 390 150)" />
  <use xlink:href="#R1" transform="rotate(144 390 150)" />
  <use xlink:href="#R1" transform="rotate(216 390 150)" />
  <use xlink:href="#R1" transform="rotate(288 390 150)" />
</svg>
```

```
@Component({
  selector: 'app-grafico-svg',
  templateUrl: './grafico-svg.component.svg',
})
export default class GraficoSvgComponent {
  fillColor = 'rgb(255, 0, 0)';
  changeColor() {
    const r = Math.floor(Math.random() * 256);
    const g = Math.floor(Math.random() * 256);
    const b = Math.floor(Math.random() * 256);
    this.fillColor = `rgb(${r}, ${g}, ${b})`;
  }
}
```

© JMA 2020. All rights reserved

FORMULARIOS

© JMA 2020. All rights reserved

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 {}
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.
 - enabled / disabled: 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: FormControlStatus, cadena con 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

© JMA 2020. All rights reserved

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	ng-pending

© JMA 2020. All rights reserved

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>
<input type="text" name="nombre" ngModel> <!-- Enlaza con el name del input -->
```
- **ngModelGroup:** Permite crear variables referencia de un grupo de controles de formulario para validar el subgrupo del formulario en su conjunto.

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

© JMA 2020. All rights reserved

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" >
```
- min: El campo numérico debe tener un valor mínimo (v12).
- max: El campo numérico debe tener un valor máximo (v12).
- 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="/^[\w-]{3,50}/" >
```
- email: El campo debe tener el formato de un correo electrónico.

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

© JMA 2020. All rights reserved

Tipos de Validaciones

- Funciones de validación en Validators:
 - required
 - requiredTrue
 - minLength
 - maxLength
 - pattern
 - email
 - min: El campo debe tener un valor mínimo
 - max: El campo debe tener un valor máximo

© JMA 2020. All rights reserved

Comprobando las validaciones

- valid e 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 de valid e invalid se puede realizar a nivel de control individual o de formulario completo pero de errors solo con controles individuales.
- Los métodos hasError y getError simplifican el acceso a los errores.
- Por defecto se marca con el atributo HTML novalidate en la etiqueta <form> y se evita que el navegador haga sus propias validaciones que choquen con las de Angular, para desactivar su inclusiva se utiliza la directiva ngNativeValidate.
<form name="miForm" ngNativeValidate >

© JMA 2020. All rights reserved

Notificación de errores

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

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

- Para resumir en un mensaje varias causas de error:

```
<output class="error" [hidden]="!nombre.hasError('minlength') &&  
!nombre.hasError('maxlength')">El nombre debe tener entre 3 y 50  
letras</output>
```

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

```
<output [hidden]="!nombre.valid || miForm.pristine" ...>
```

© JMA 2020. All rights reserved

Enviar formulario

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

```
<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 && this.isValid(miForm.value)) {  
        ...  
    } else {  
        alert("Hay datos inválidos");  
    }  
}
```

© JMA 2020. All rights reserved

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;
  // ...
}
```

© JMA 2020. All rights reserved

Validaciones personalizadas

```
export function NIFValidator(): ValidatorFn {
  return (control: AbstractControl): { [key: string]: any } | null => {
    if (!control.value) { return null; }
    const err = { nif: { invalidFormat: true, invalidChar: true } };
    if (/^\d{1,8}\w$/.test(control.value)) {
      const letterValue = control.value.substr(control.value.length - 1);
      const numberValue = control.value.substr(0, control.value.length - 1);
      err.nif.invalidFormat = false;
      return letterValue.toUpperCase() === 'TRWAGMYFPDXBNJZSQVHLCKE'.charAt(numberValue % 23) ? null : err;
    } else { return err; }
  };
}
@Directive({
  selector: '[nif][formControlName],[nif][formControl],[nif][ngModel]',
  providers: [{ provide: NG_VALIDATORS, useExisting: NIFValidatorDirective, multi: true }]
})
export class NIFValidatorDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    return NIFValidator()(control);
  }
}
<input type="text" name="dni" [(ngModel)]="model.dni" #dni="ngModel" nif>
<div *ngIf="dni?.errors?.nif">No es un NIF valido.</div>
```

© JMA 2020. All rights reserved

Validaciones personalizadas

```
@Directive({
  selector: '[type][formControlName],[type][formControl],[type][ngModel]',
  providers: [
    { provide: NG_VALIDATORS, useExisting: forwardRef(() => TypeValidatorDirective), multi: true }
  ]
})
export class TypeValidatorDirective implements Validator {
  constructor(private elem: ElementRef) {}
  validate(control: AbstractControl): ValidationErrors | null {
    const valor = control.value;
    if (valor) {
      const dom = this.elem.nativeElement;
      if (dom.validity) { // dom.checkValidity();
        return dom.validity.typeMismatch ? { 'type': dom.validationMessage } : null;
      }
    }
    return null;
  }
}

<input type="url" id="foto" name="foto" [(ngModel)]="elemento.foto" #foto="ngModel">
<output class="error" [hidden]="!foto.errors?.type">{{foto.errors.type}}</output>
```

© JMA 2020. All rights reserved

Validaciones personalizadas

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

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

    if (valor) {
      return (!cntrlBind || valor !== cntrlBind.value) ? { 'equalsTo': `${valor} <> ${cntrlBind?.value}` } : null;
    }
    return null;
  }
}

<input type="password" ... #pwd="ngModel">
<input type="password" name="valPwd" id="valPwd" [(ngModel)]="valPwd" #valPwd="ngModel" equalsTo="pwd">
<div *ngIf="valPwd?.errors?.equalsTo">Las dos claves deben ser iguales.</div>
```

© JMA 2020. All rights reserved

Formularios Reactivos

- Los 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 Angular de control de formulario (`AbstractControl`) en la clase del 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 en el formulario y reaccionar ante esos cambios.

© JMA 2020. All rights reserved

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 se decide utilizarlo.

© JMA 2020. All rights reserved

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}"

© JMA 2020. All rights reserved

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 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 crearlos.

© JMA 2020. All rights reserved

Creación del formulario

- Hay que definir en la clase del componente una propiedad pública 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));
```

© JMA 2020. All rights reserved

Creación del formulario

- Para 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.

© JMA 2020. All rights reserved

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),  
});
```

© JMA 2020. All rights reserved

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.map(  
    item => new FormGroup({ role: new FormControl(item.role, Validators.required) })  
));  
this.miForm = new FormGroup({  
    user: new FormControl(""),  
    password: // ...  
    roles: fa  
});
```

© JMA 2020. All rights reserved

Agregaciones

- Para añadir un nuevo elemento a la agregación:

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

- Para eliminar un elemento de la agregación:

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

© JMA 2020. All rights reserved

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  
});
```

© JMA 2020. All rights reserved

Validaciones personalizadas

- Para crear un validación personalizada se crea una factoría (función que devuelve una función) que recibe un numero arbitrario de parámetros y devuelve una función de validación configurada (función que se invocara en las validaciones). La firma de la función de validación devuelta es:
 - `(control: AbstractControl): { [key: string]: any } | ValidationErrors | null`
- La función recibe el `AbstractControl` al que se aplica la validación, usando la propiedad `control.value` se accede al valor a validar. No puede recibir mas argumentos pero puede utilizar los parámetros de la factoría vía clausura.
- Devuelve un valor nulo si el valor del control es válido o un objeto de error de validación. El objeto de error de validación normalmente tiene una propiedad cuyo nombre es la clave de validación y cuyo valor no debe ser falsify y puede ser escalar o un diccionario arbitrario de valores que se pueden insertar en un mensaje de error. El resultado se mezcla con los resultados del resto de validaciones en la colección `AbstractControl.errors`, donde las claves sirven para identificar los errores producidos. Por convenio el valor vacío del control se considera siempre valido para combinar con `required` cuando no lo sea.
- Los validadores asíncronos deben devolver una promesa u observable que luego emita el nulo u objeto de error de validación. En el caso de un observable, el observable debe completarse, momento en el que el formulario utiliza el último valor emitido para la validación.

© JMA 2020. All rights reserved

Validaciones personalizadas

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

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

```
export const passwordMatchValidator: ValidatorFn = (control: AbstractControl): ValidationErrors | null => {
  const pwdOrg = control.get('passwordValue');
  const pwdCopy = control.get('passwordConfirm');
  return pwdOrg && pwdCopy && pwdOrg.value === pwdCopy.value ? null : {
    passwordMatchValidator: true };
};
```

© JMA 2020. All rights reserved

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 permite actualizaciones parciales y 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();
```

© JMA 2020. All rights reserved

Enlace de datos

- Para modificar el estado de un FormGroup o FormControl:

```
const control = miForm.get('user')
control.markAsTouched();
control.markAsUntouched();
control.markAsDirty();
control.markAsPristine();
```
- Para cambiar los validadores:

```
const validators: ValidatorFn[] = [...];
this.miForm.setValidators(validators);
this.miForm.get('user').setValidators(validators);
```
- Para borrar los validadores del formulario o control:

```
this.miForm.clearValidators();
this.miForm.get('user').clearValidators();
```

© JMA 2020. All rights reserved

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  
  });
```

© JMA 2020. All rights reserved

FormBuilder

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

```
constructor(protected fb: FormBuilder) {}
```
- Mediante una estructura se crea el formulario:

```
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: this.fb.array(this.model.roles.map(item => this.fb.group({ role: [item.role,  
    Validators.required] })))  
});
```

© JMA 2020. All rights reserved

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>
```

© JMA 2020. All rights reserved

Mostrar errores

- Errores en los controles del formulario:

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

© JMA 2020. All rights reserved

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}}
```

© JMA 2020. All rights reserved

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 }}
```

© JMA 2020. All rights reserved

Comprobación estricta de tipos

- En las versiones anteriores de Angular Reactive Forms el valor emitido por el formulario era de tipo any, el marco no tenía mucha información disponible sobre los tipos de cada control de formulario por lo que no disponía de seguridad de tipos (comprobación estricta de tipos).
- Pero a partir de Angular 14 y en adelante, Angular ha agregado la seguridad de tipos completa integrada a Reactive Forms, lo que significa que se pueden recibir mensajes de error útiles y permite el autocompletado cuando se trabaja con valores de formulario.
- Se han reemplazado las clases FormControl, FormGroup y FormArray por sus versiones genéricas que permiten establecer explícitamente o inferir el tipo del valor de inicialización.
- Han aparecido las versiones UntypedFormControl, UntypedFormGroup y UntypedFormArray sin tipo (tipo any).
- Todos los FormControl permiten la nulabilidad salvo que se utilice en el constructor la opción {nonNullable: true}, en cuyo caso .reset restablece a su valor inicial en lugar de null.
- En los FormArray, el parámetro de tipo corresponde al tipo de cada control interno. Si se desea tener varios tipos de elementos diferentes se debe usar un UntypedFormArray.

© JMA 2020. All rights reserved

Comprobación estricta de tipos

- Algunos FormGroup (grupos dinámicos) tienen controles que pueden o no estar presentes, que se pueden agregar y eliminar en tiempo de ejecución. En estos casos se debe definir un interface o tipo usando campos opcionales con las claves. Las operaciones .addControl y .removeControl están restringidas a los opcionales.
- Cuando las claves no se conocen de antemano, la clase FormRecord permite añadirlos dinámicamente siempre que todos los controles sean del mismo tipo.

```
const addresses = new FormRecord<FormControl<string | null>>({});  
addresses.addControl('Andrew', new FormControl('2340 Folsom St'));
```
- Si se necesita un FormGroup que sea tanto dinámico (abierto) como heterogéneo (los controles son de diferentes tipos) solo se puede usar UntypedFormGroup.
- La clase FormBuilder agrega los controles como nullables. Se ha añadido .nonNullable como una forma abreviada de especificar {nonNullable: true} en cada control y eliminar un texto repetitivo significativo de formularios grandes que no aceptan valores NULL
- También se puede inyectar como servicio usando el nombre NonNullableFormBuilder.

© JMA 2020. All rights reserved

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 asíncronía de los formularios basados en plantillas también complica las pruebas unitarias. Se debe colocar el bloque de prueba en waitForAsync() o en fakeAsync() para evitar buscar valores aún no disponibles en el formulario. Con los formularios reactivo, todo está disponible tal y como se espera que sea.

© JMA 2020. All rights reserved

DIRECTIVAS

© JMA 2020. All rights reserved

Introducción

- 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 debe manipular el árbol DOM es 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.
- Las directivas son clases decoradas con @Directive.

© JMA 2020. All rights reserved

Características de las directivas

- Las directivas se clasifican en dos tipos:
 - directivas estructurales: que alteran el diseño mediante la adición, eliminación y sustitución de elementos en DOM.
 - directivas atributos: que alteran la apariencia o el comportamiento de un elemento existente.
- Prefijos en Directivas
 - El equipo de Angular recomienda que las directivas tengan siempre un prefijo de forma que no choquen con otras directivas creadas por otros desarrolladores o con actuales o futuras etiquetas de HTML. Por eso las directivas de Angular empiezan siempre por "ng".
- Nombre
 - El nombre de las directivas utiliza la notación camelCase sigue el formato de los identificadores en JavaScript. Es el nombre que se usa en la documentación de Angular y el que se usa cuando se crea una nueva directiva.

© JMA 2020. All rights reserved

Registrar

- Globalmente:

```
@NgModule({  
  ...  
  declarations: [ ... , MyDirective ],  
  ...  
  exports: [ MyDirective ] // En los módulos de características  
})  
export class AppModule { }
```
- En el contenedor:

```
@Component({  
  ...  
  entryComponents: [ ... , MyDirective ],  
  ...  
})  
export class AppComponent { ... }
```

© JMA 2020. All rights reserved

@Directive

- **selector:** Selector CSS que indica a Angular que debe ejecutar la directiva cuando se encuentra un elemento con ese nombre en el HTML.
- **providers:** Lista de los proveedores disponibles para esta directiva.
- **inputs:** Lista de nombres de las propiedades que se enlazan a datos con las entradas del componente.
- **outputs:** Lista de los nombres de las propiedades de la clase que exponen a los eventos de salida que otros pueden suscribirse.
- **exportAs:** nombre bajo el cual se exporta la instancia de la directiva para asignarlas en las variables referencia en las plantillas.
- **host:** enlace de eventos, propiedades y atributos del elemento host con propiedades y métodos de la clase.
- **queries:** configura las consultas que se pueden injectar en el componente

© JMA 2020. All rights reserved

Directiva personalizada

```
import { Directive, Input, Output, HostListener, EventEmitter, HostBinding } from '@angular/core';

@Directive({ selector: '[winConfirm]' })
export class WindowConfirmDirective {
  @Input() winConfirmMessage = '¿Seguro?';
  @Output() winConfirm: EventEmitter<any> = new EventEmitter();
  @HostBinding('class.pressed') isPressed: boolean = false;

  @HostListener('click', ['$event'])
  confirmFirst() {
    if (window.confirm(this.winConfirmMessage)) {
      this.winConfirm.emit(null);
    }
  }
  @HostListener('mousedown') hasPressed() { this.isPressed = true; }
  @HostListener('mouseup') hasReleased() { this.isPressed = false; }
}

@Directive({ selector: '[show]' })
export class ShowDirective {
  @HostBinding('hidden') hidden: boolean = false;
  @Input('myShow') set show(value: boolean) { this.hidden = !value; }
}
<button (winConfirm)="vm.delete(p.id)" winConfirmMessage="¿Estás seguro?">Borrar</button>
```

© JMA 2020. All rights reserved

Selector

- El selector CSS desencadena la creación de instancias de una directiva.
- Angular sólo permite directivas que se desencadenen sobre los selectores CSS que no crucen los límites del elemento sobre el que están definidos.
- El selector se puede declarar como:
 - element-name: nombre de etiqueta.
 - .class: nombre de la clase CSS que debe tener definida la etiqueta.
 - [attribute]: nombre del atributo que debe tener definida la etiqueta.
 - [attribute=value]: nombre y valor del atributo definido en la etiqueta.
 - [attribute][ngModel]: nombre de los atributos que debe tener definida la etiqueta, condicionado a que aparezca el segundo atributo.
 - :not(sub_selector): Seleccionar sólo si el elemento no coincide con el sub_selector.
 - selector1, selector2: Varios selectores separados por comas.

© JMA 2020. All rights reserved

Atributos del selector

- Los atributos del selector de una directiva se comportan como los atributos de las etiquetas HTML, permitiendo personalizar y enlazar a la directiva desde las plantillas.
- Propiedades de entrada

```
@Input() init: string;  
<my-comp myDirective [init]="1234"></my-comp>
```
- Eventos de salida

```
@Output() updated: EventEmitter<any> = new EventEmitter();  
this.updated.emit(value);  
<myDirective (updated)="onUpdated($event)"></myDirective>
```
- 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>();
```

© JMA 2020. All rights reserved

Directiva atributo con valor

- La presencia de la directiva como etiqueta o como atributo dispara la ejecución de la misma.
- En algunos casos, con la notación atributo, es necesario asignar un valor que la personalice.
- Como propiedad de entrada:

```
@Input('myDirective') valor: string;
```
- Inyectado en el constructor:

```
constructor(@Attribute('myDirective') public valor: string) {}
```
- Se enlaza como cualquier otra propiedad:

```
<div [myDirective]="myValue" (myDirective)="click()">
```

© JMA 2020. All rights reserved

Vinculación de propiedades

- Podemos usar directivas de atributos que afecten el valor de las propiedades en el nodo del host usando el decorador `@HostBinding`.
- El decorador `@HostBinding` permite programar un valor de propiedad en el elemento host de la directiva.
- Funciona de forma similar a una vinculación de propiedades definida en una plantilla, excepto que se dirige específicamente al elemento host.
- La vinculación se comprueba para cada ciclo de detección de cambios, por lo que puede cambiar dinámicamente si se desea.

```
@HostBinding('class.pressed') isPressed: boolean = false;  
@HostListener('mousedown') hasPressed() { this.isPressed = true; }  
@HostListener('mouseup') hasReleased() { this.isPressed = false; }
```

© JMA 2020. All rights reserved

Eventos del DOM

- Se podría llegar al DOM con JavaScript estándar y adjuntar manualmente los controladores de eventos pero hay al menos tres problemas con este enfoque:
 - Hay que asociar correctamente el controlador.
 - El código debe desasociar manualmente el controlador cuando la directiva se destruye para evitar pérdidas de memoria.
 - Interactuar directamente con API DOM no es una buena práctica.
- Mediante el decorador `@HostListener` de `@angular/core` se puede asociar un evento a un método de la clase:

```
@HostListener('mouseenter', ['$event'])  
onMouseEnter(e: any) { ... }
```
- Se pueden asociar eventos de `window`, `document` y `body`:

```
@HostListener('document:click', ['$event'])  
handleClick(event: Event) { ... }
```

© JMA 2020. All rights reserved

Directivas estructurales

- La gran diferencia con las directivas atributo es que, debido a la naturaleza de las directivas estructurales vinculadas a una plantilla, tenemos acceso a TemplateRef, un objeto que representa la etiqueta de plantilla a la que se adjunta la directiva (elementRef), y a ViewContainerRef, que gestiona las vistas del contenido.

constructor(

```
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef) { }
```

- Una directiva estructural sencilla crea una vista incrustada para el <ng-template> generado por Angular e inserta la vista en el contenedor de vistas adyacente al elemento host que contiene la directiva.

© JMA 2020. All rights reserved

Directivas estructurales

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({ selector: '[myUnless]' })
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set myUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}

<div *myUnless="isValid" >
```

© JMA 2020. All rights reserved

Contenedores de vistas

- Los ViewContainer son contenedores en los que se pueden adjuntar una o más vistas.
- Las vistas representan algún tipo de diseño que se va a representar y el contexto en el que se debe procesar.
- Los ViewContainer están anclados a los componentes y son responsables de generar su salida, de modo que esto significa que el cambio de las vistas que se adjuntan al ViewContainer afecta a la salida final del componente.
- Se pueden adjuntar dos tipos de vistas a un contenedor de vistas: vistas de host que están vinculadas a un componente y vistas incrustadas vinculadas a una plantilla.
- Las directivas estructurales interactúan con las plantillas por lo que utilizan vistas incrustadas.

© JMA 2020. All rights reserved

Contenedores de vistas

`createEmbeddedView(templateRef: TemplateRef<C>, context?: C, index?: number): EmbeddedViewRef<C>`

- Instancia una vista incrustada basada en `templateRef`, asociada a un contexto y la inserta en el contenedor en la posición especificada por el índice.
- Si no se especifica el índice, la nueva vista se insertará como la última vista del contenedor.
`this.viewContainer.createEmbeddedView(this.templateRef);`
- Se puede adjuntar un objeto de contexto que debe ser un objeto de claves/valor donde las claves estarán disponibles para la vinculación por las declaraciones de plantilla local (la clave `$implicit` en el objeto de contexto establecerá el valor predeterminado para las claves no existentes).

`clear(): void`

- Destruye todas las vistas del contenedor.
`this.viewContainer.clear();`

© JMA 2020. All rights reserved

Acceso directo al DOM

- El servicio ElementRef permite el acceso directo al elemento DOM a través de su propiedad nativeElement.
- Solo se debe usar como el último recurso cuando se necesita acceso directo a DOM:
 - puede hacer la aplicación más vulnerables a los ataques XSS.
 - crea acoplamiento entre la aplicación y las capas de renderizado imposibilitando su separación para el uso Web Workers.
- El servicio Renderer2 proporciona un API que se puede utilizar con seguridad incluso cuando el acceso directo a elementos nativos no es compatible.

```
@Directive({ selector: '[myShadow]' })
export class ShadowDirective {
  constructor(el: ElementRef, renderer: Renderer2) {
    //el.nativeElement.style.boxShadow = '10px 10px 5px #888888';
    renderer.setStyle(el.nativeElement, 'box-shadow', '10px 10px 5px #888888');
  }
}
```

© JMA 2020. All rights reserved

API de composición de directivas (v15)

- La API de composición de directivas permite aplicar directivas al elemento host de un componente desde dentro del propio componente o directiva, siendo una poderosa estrategia de reutilización de código. La API de composición de directivas solo funciona con directivas independientes.
- La composición de directivas se aplica agregando una propiedad hostDirectives al decorador @Directive (o @Component). Acepta una serie de directivas independientes y las aplicará en el componente host.

```
@Directive({})
export class Menu {}  
  
@Directive({})
export class Tooltip {}  
  
// MenuWithTooltip puede componer comportamientos a partir de otras directivas
@Directive({
  hostDirectives: [Tooltip, Menu],
})
export class MenuWithTooltip {}
```

© JMA 2020. All rights reserved

API de composición de directivas (v15)

- Las entradas y salidas de las directivas de host no se incluyen en la API de su componente de forma predeterminada. Se pueden incluir explícitamente las entradas y salidas en la API de su componente expandiendo la entrada (se pueden crear alias de entradas y salidas) en la propiedad hostDirectives:

```
@Component({
  selector: 'admin-menu',
  template: 'admin-menu.html',
  hostDirectives: [
    {
      directive: MenuBehavior,
      inputs: ['menuId'],
      outputs: ['menuClosed: closed'],
    },
  ],
})
export class AdminMenu {}  
<admin-menu menuId="top-menu" (closed)="logMenuClosed()">
```

© JMA 2020. All rights reserved

Validación personalizada

- A través de las directivas se pueden implementar validaciones personalizadas para los formularios dirigidos por plantillas.
- Deben implementar el método validate del interface Validator.
validate(control: AbstractControl): { [key: string]: any } { ... }
- El método recibe el AbstractControl al que se aplica la validación, usando la propiedad control.value se accede al valor a validar.
- El método devuelve null si el valor valido (el valor vacío se considera siempre valido salvo en el required) o un array de claves/valor.
- El resultado se mezcla con los resultados del resto de validaciones en la colección AbstractControl.errors, donde las claves sirven para identificar los errores producidos.
- La validación debería implementarse mediante la invocación de una función externa con la misma firma que el método para favorecer su reutilización en los formularios reactivos.

© JMA 2020. All rights reserved

Validación personalizada

- El selector debería incluir los atributos [formControlName], [formControl] y [ngModel] para limitar el ámbito de selección.
- Angular, como mecanismo interno para ejecutar los validadores en un control de formulario, mantiene un proveedor múltiple de dependencias (array de dependencias) denominado NG_VALIDATORS. Utilizando la propiedad multi: true se indica que acumule la dependencia en un array (proveedor múltiple) en vez de sustituir el anterior.
- Todos los validadores predefinidos ya se encuentran agregados a NG_VALIDATORS. Así que cada vez que Angular instancia un control de formulario, para realizar la validación, inyecta las dependencias de NG_VALIDATORS, que es la lista de todos los validadores, y los ejecuta uno por uno con el control instanciado.
- Es necesario registrar los nuevos validadores en NG_VALIDATORS como providers en @Directive.

© JMA 2020. All rights reserved

Validación personalizada

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

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

@Directive({
  selector: '[naturalNumber][formControlName],[naturalNumber][formControl],[naturalNumber][ngModel]',
  providers: [{ provide: NG_VALIDATORS, useExisting: NaturalNumberValidatorDirective, multi: true }]
})
export class NaturalNumberValidatorDirective implements Validator {
  validate(control: AbstractControl): { [key: string]: any } {
    if (control.value) {
      return naturalNumberValidator()(control);
    }
    return null;
  }
}

<input type="text" name="edad" id="edad" [(ngModel)]="model.edad" #edad="ngModel" naturalNumber>
<output class="errorMsg" [hidden]="!edad?.errors?.naturalNumber">No es un número entero positivo.</output>
```

© JMA 2020. All rights reserved

Validaciones personalizadas

```
export function NIFValidator(): ValidatorFn {
  return (control: AbstractControl): { [key: string]: any } | null => {
    if (!control.value) { return null; }
    const err = { nif: { invalidFormat: true, invalidChar: true } };
    if (/^[\d]{1,8}w$/.test(control.value)) {
      const letterValue = control.value.substr(control.value.length - 1);
      const numberValue = control.value.substr(0, control.value.length - 1);
      err.nif.invalidFormat = false;
      return letterValue.toUpperCase() === 'TRWAGMYFPDXBNJZSQVHLCKE'.charAt(numberValue % 23) ? null : err;
    } else { return err; }
  };
}

@Directive({
  selector: '[nif][formControlName],[nif][formControl],[nif][ngModel]',
  providers: [{ provide: NG_VALIDATORS, useExisting: NIFValidatorDirective, multi: true }]
})
export class NIFValidatorDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    return NIFValidator()(control);
  }
}
<input type="text" name="dni" [(ngModel)]="model.dni" #dni="ngModel" nif>
<div *ngIf="dni?.errors?.nif">No es un NIF valido.</div>
```

© JMA 2020. All rights reserved

Validaciones personalizadas

```
@Directive({
  selector: '[type][formControlName],[type][formControl],[type][ngModel]',
  providers: [
    { provide: NG_VALIDATORS, useExisting: forwardRef(() => TypeValidatorDirective), multi: true }
  ]
})
export class TypeValidatorDirective implements Validator {
  constructor(private elem: ElementRef) {}
  validate(control: AbstractControl): ValidationErrors | null {
    const valor = control.value;
    if (valor) {
      const dom = this.elem.nativeElement;
      if (dom.validity) { // dom.checkValidity();
        return dom.validity.typeMismatch ? { 'type': dom.validationMessage } : null;
      }
    }
    return null;
  }
}
<input type="url" id="foto" name="foto" [(ngModel)]="elemento.foto" #foto="ngModel">
<output class="error" [hidden]="!foto.errors?.type">{{foto.errors.type}}</output>
```

© JMA 2020. All rights reserved

Validaciones personalizadas

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

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

    if (valor) {
      return (!cntrlBind || valor !== cntrlBind.value) ? { 'equalsTo': `'$valor' <> ${cntrlBind?.value}` } : null;
    }
    return null;
  }
}

<input type="password" ... #pwd="ngModel">
<input type="password" name="valPwd" id="valPwd" [(ngModel)]="valPwd" #valPwd="ngModel" equalsTo="pwd">
<div *ngIf="valPwd?.errors?.equalsTo">Las dos claves deben ser iguales.</div>
```

© JMA 2020. All rights reserved

Shared Component

- Aunque un componente es técnicamente una directiva-con-plantilla, conceptualmente son dos elementos completamente diferentes.
- Los componentes se pueden clasificar como:
 - Componentes de aplicación: Son componentes íntimamente ligados a la aplicación cuya existencia viene determinada por la existencia de la aplicación. Estos componentes deben favorecer el desacoplamiento de entre la plantilla y la clase del componente.
 - Componentes compartidos: Son componentes de bajo nivel, que existen para dar soporte a los componentes de aplicación pero son independientes de aplicaciones concretas. Estos componentes requieren un mayor control sobre la plantilla por los que el acoplamiento será mayor, así como la dependencia entre ellos.

© JMA 2020. All rights reserved

Dynamic Component

- Las plantillas de los componentes no siempre son fijas. Es posible que una aplicación necesite cargar nuevos componentes en tiempo de ejecución.
- Angular dispone del servicio ComponentFactoryResolver para cargar componentes dinámicamente.

```
@Component({
  selector: 'dynamyc-template',
  template: `<ng-template my-host></ng-template>`
})
export class DynamicComponent implements AfterViewInit {
  @ViewChild(MyHostDirective) myHost: MyHostDirective;
  constructor(private componentFactoryResolver: ComponentFactoryResolver) { }
  ngAfterViewInit() { this.loadComponent(); }
  loadComponent() {
    let componentFactory = this.componentFactoryResolver.resolveComponentFactory(DemoTemplateComponent);
    let viewContainerRef = this.myHost.viewContainerRef;
    viewContainerRef.clear();
    let componentRef = viewContainerRef.createComponent(componentFactory);
  }
}
```

© JMA 2020. All rights reserved

ng-container

- 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 elementos 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>
```

© JMA 2020. All rights reserved

*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>  
public componente: Type<any> = MyComponent;
```

© JMA 2020. All rights reserved

*ngTemplateOutlet

- Inserta una vista incrustada desde un TemplateRef preparado

```
@Component({  
  selector: 'ng-template-outlet-example',  
  template: `  
    <ng-container *ngTemplateOutlet="greet"></ng-container>  
    <hr>  
    <ng-container *ngTemplateOutlet="eng; context: myContext"></ng-container>  
    <hr>  
    <ng-container *ngTemplateOutlet="svk; context: myContext"></ng-container>  
    <hr>  
  
    <ng-template #greet><span>Hello</span></ng-template>  
    <ng-template #eng let-name><span>Hello {{name}}!</span></ng-template>  
    <ng-template #svk let-person="localSk"><span>Ahoj {{person}}!</span></ng-template>  
  
  `}  
)  
export class NgTemplateOutletExample {  
  myContext = {$implicit: 'World', localSk: 'Svet'};  
}
```

© JMA 2020. All rights reserved

ngTemplateOutletContext

- Enlaza una vista incrustada ngTemplateOutlet con un contexto de datos

```
@Component({
  selector: 'demo-template',
  template: `
    <ng-template #templateRef let-label="label" let-url="url">
      <div><a href="{{url}}>{{label}}*</a></div>
    </ng-template>
    <h1>Menu</h1>
    <div [ngTemplateOutlet]="templateRef" [ngTemplateOutletContext]="menu[0]"></div>
    <div [ngTemplateOutlet]="templateRef" [ngTemplateOutletContext]="menu[1]"></div>
  `
})
export class DemoTemplateComponent {
  menu:any = [
    { "id": 1, "label": "AngularJS", "url": "http://angularjs.org" },
    { "id": 2, "label": "Angular", "url": "http://angular.io" }
  ];
}
```

© JMA 2020. All rights reserved

ng-content

- La directiva <ng-content> es un elemento de transclusión (trasladar e incluir) del contenido de la etiqueta del componente a la plantilla.

```
<card [header]="">Este es el contenido a transcluir que puede contener otras etiquetas y componentes</card>
```

```
@Component({
  selector: 'card',
  template: `
    <div class="card">
      <H1>{{ header }}</H1>
      <ng-content></ng-content>
    </div>`,
})
export class CardComponent {
  @Input() header: string = 'this is header';
```

© JMA 2020. All rights reserved

ng-content

- <ng-content> acepta un atributo select que permite definir un selector CSS que indica el contenido que se muestra en la directiva.
- Con el uso del atributo select se puedan definir varios ng-content, uno por cada tipo de contenido.
- Selector asociado a un atributo de HTML:
`<ng-content select="[titulo]"></ng-content>`
- Selector asociado a un class de CSS:
`<ng-content select=".cuerpo"></ng-content>`
- Contenido:
`<card [header]=""Hola mundo"">
 <div titulo <div class="cuerpo" componentes</div>
</card>`

© JMA 2020. All rights reserved

Controles de formulario personalizados

- Para que un componente pueda interactuar con ngModel como un control de formulario debe implementar el interfaz ControlValueAccessor.
- La interfaz ControlValueAccessor se encarga de:
 - Escribir un valor desde el modelo de formulario en la vista / DOM
 - Informar a otras directivas y controles de formulario cuando cambia la vista / DOM
- Los métodos a implementar son:
 - writeValue(obj: any): void
 - registerOnChange(fn: any): void
 - registerOnTouched(fn: any): void
 - setDisabledState(isDisabled: boolean)?: void
- Hay que registrar el nuevo control:
`{ provide: NG_VALUE_ACCESSOR, useExisting: MyControlComponent, multi: true }`

© JMA 2020. All rights reserved

XSS

- Los scripts de sitios cruzados (XSS) permiten a los atacantes injectar código malicioso en las páginas web. Tal código puede, por ejemplo, robar datos de usuario (en particular, datos de inicio de sesión) o realizar acciones para suplantar al usuario. Este es uno de los ataques más comunes en la web.
- Para bloquear ataques XSS, se debe evitar que el código malicioso entre al DOM (Modelo de Objetos de Documento). Por ejemplo, si los atacantes pueden engañarte para que insertes una etiqueta `<script>` en el DOM, pueden ejecutar código arbitrario en tu sitio web.
- El ataque no está limitado a las etiquetas `<script>`, muchos elementos y propiedades en el DOM permiten la ejecución del código, por ejemplo, ``. Si los datos controlados por el atacante ingresan al DOM, se esperan vulnerabilidades de seguridad.``
- Para bloquear sistemáticamente los errores XSS, por defecto Angular trata todos los valores como no confiables.
- Cuando se inserta un valor en el DOM desde una plantilla, mediante propiedad, atributo, estilo, enlace de clase o interpolación, Angular desinfecta y escapa de los valores no confiables.
- Las plantillas Angular son las mismas que las del código ejecutable: se confía en que el HTML, los atributos y las expresiones vinculantes (pero no los valores vinculados) en las plantillas son seguros.
- Esto significa que las aplicaciones deben evitar que los valores que un atacante puedan controlar puedan convertirse en el código fuente de una plantilla.
- Nunca genere código fuente de plantilla concatenando la entrada del usuario y de las plantillas.
- Para evitar estas vulnerabilidades, debe usarse el compilador de plantilla sin conexión, también conocido como inyección de plantilla

© JMA 2020. All rights reserved

Contextos de saneamiento y seguridad

- La desinfección es la inspección de un valor que no es de confianza, convirtiéndolo en un valor que es seguro insertar en el DOM. En muchos casos, la desinfección no cambia un valor en absoluto. La desinfección depende del contexto: un valor inofensivo en CSS es potencialmente peligroso en una URL.
- Angular define los siguientes contextos de seguridad:
 - HTML se usa cuando se interpreta un valor como HTML, por ejemplo, cuando se vincula a `innerHTML`.
 - El estilo se usa cuando se vincula CSS a la propiedad `style`.
 - La URL se usa para propiedades de URL como `<a href>`.
 - La URL es una URL de recurso que se cargará y ejecutará como código, por ejemplo, en ``.
- Angular desinfecta los valores que no son de confianza para HTML, estilos y URL. La desinfección de URL de recursos no es posible porque contienen código arbitrario. En modo de desarrollo, Angular imprime una advertencia en consola cuando tiene que cambiar un valor durante la desinfección.

© JMA 2020. All rights reserved

DomSanitizer

- A veces, las aplicaciones realmente necesitan incluir código ejecutable, mostrar un <iframe> desde alguna URL o construir una URL potencialmente peligrosas. Para evitar la desinfección automática en cualquiera de estas situaciones, puede decirle a Angular que inspeccionó un valor, verificó cómo se generó y se aseguró de que siempre sea seguro. El peligro está en confiar en un valor que podría ser malicioso, se está introduciendo una vulnerabilidad de seguridad en su aplicación.
- Para marcar un valor como confiable, hay que injectar el servicio DomSanitizer y llamar a uno de los siguientes métodos para convertir la entrada del usuario en un valor confiable:
 - bypassSecurityTrustHtml
 - bypassSecurityTrustScript
 - bypassSecurityTrustStyle
 - bypassSecurityTrustUrl
 - bypassSecurityTrustResourceUrl
- Un valor es seguro dependiendo del contexto, es necesario elegir el contexto correcto para su uso previsto del valor.

© JMA 2020. All rights reserved

CONCEPTOS AVANZADOS

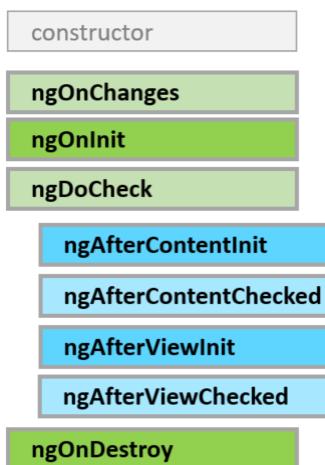
© JMA 2020. All rights reserved

Conceptos avanzados

- Arquitectura avanzada y mejores prácticas
- Ciclo de vida y detección de cambios
- Optimización y rendimiento
- Internacionalización (I18N)
- Publicación de módulos con NPM
- Migración desde AngularJS a Angular
- Módulo ngUpgrade: Aplicaciones hibridas AngularJS y Angular 2+

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Ciclo de vida

- Las instancias de componentes y directivas tienen un ciclo de vida ya que Angular las crea, las actualiza y las destruye. Los desarrolladores pueden aprovechar los momentos clave en ese ciclo de vida mediante la implementación de una o más de las interfaces de enlace de ciclo de vida en la biblioteca core de Angular denominados ganchos.
- Las interfaces son opcionales para desarrolladores de JavaScript y de Typescript desde una perspectiva puramente técnica. El lenguaje JavaScript no tiene interfaces. Angular no puede ver las interfaces de TypeScript en tiempo de ejecución porque desaparecen del JavaScript transpilado.
- Afortunadamente, no son imprescindibles. No es necesario agregar las interfaces para beneficiarse de los propios ganchos.
- En cambio, Angular inspecciona las clases de directivas y componentes buscando los métodos del interfaz y los invoca en caso de estar definidos.
- No obstante, es una buena práctica agregar interfaces a las clases de directivas y componentes de TypeScript para beneficiarse de las herramientas de los entornos de desarrollo.

© JMA 2020. All rights reserved

OnChanges

- Angular llama al método `ngOnChanges()` cada vez que detecta cambios en las propiedades de entrada del componente (o directiva).
- El método `ngOnChanges(changes: SimpleChanges)` recibe un objeto que asigna cada nombre de propiedad cambiado a un objeto `SimpleChange` que contiene el valor actual y anterior de propiedad que ha cambiado.

```
let cur = changes[propName].currentValue;
let prev = changes[propName].previousValue;
```
- Angular solo llama a `ngOnChanges()` cuando cambia el valor de la propiedad de entrada. En los objetos el valor es la referencia por lo que los cambios interiores del objetos en el mismo objeto no se detectan.
- Otra posibilidad de interceptar cambios en una propiedad de entrada es con un setter:

```
@Input() set name(value: string) { ... }
```

© JMA 2020. All rights reserved

OnInit

- Se debe usar `ngOnInit()` para:
 - realizar inicializaciones complejas poco después de la construcción.
 - configurar el componente después de que Angular establezca las propiedades de entrada.
- Los constructores deben ser ligeros y seguros de construir. No deben solicitarse datos en un constructor de componentes. No debe de preocuparnos que un componente nuevo intente contactar a un servidor remoto cuando se haya creado para probar o antes de que se decida mostrarlo. Los constructores no deberían hacer más que establecer las variables locales iniciales en valores simples.
- Un `ngOnInit()` es un buen lugar para que un componente obtenga sus datos iniciales. Es donde pertenece la lógica pesada de inicialización.
- Hay que destacar que las propiedades de entrada de datos de una directiva no se establecen hasta después de la construcción. Eso es un problema si se quiere inicializar el componente en el constructor basándose en dichas propiedades. En cambio, ya se habrán establecido cuando se ejecute `ngOnInit()`.
- Aunque para acceder a esas propiedades suele ser preferible el método `ngOnChanges()` dado que se ejecuta antes que `ngOnInit()` y tantas veces como es necesario. Solo se llama a `ngOnInit()` una vez.

© JMA 2020. All rights reserved

DoCheck

- Después de cada ciclo de detección de cambios se invoca `ngDoCheck()` para detectar y actuar sobre los cambios que Angular no atrapa por sí mismo.
- Debe inspeccionar ciertos valores de interés, capturando y comparando su estado actual con los valores anteriores cacheados.
- Si bien `ngDoCheck()` puede detectar cambios que Angular pasó por alto, tiene un costo aterrador dado que se llama con una frecuencia enorme, después de cada ciclo de detección de cambio, sin importar dónde ocurrió el cambio
- La mayoría de estas comprobaciones iniciales se desencadenan por la primera representación de Angular de datos no relacionados en otro lugar de la página. El simple hecho de pasar a otro `<input>` dispara una llamada y la mayoría de las llamadas no revelan cambios pertinentes.
- La implementación debe ser muy ligera o la experiencia de usuario se resiente.

© JMA 2020. All rights reserved

AfterContent

- `ngAfterContentInit()` y `ngAfterContentChecked()` se invocan después de que Angular proyecta contenido externo en el componente, `textContent` del componente.
- La proyección de contenido es una forma de importar contenido HTML desde fuera del componente e insertar ese contenido en la plantilla del componente en el lugar designado (anteriormente transclusión).
`<app-demo>Mi contenido</app-demo>`
- Nunca debe colocarse contenido entre las tag de un componente a menos que se tenga la intención de proyectar ese contenido en el componente.
- Actúan basándose en valores cambiantes en un elemento secundario de contenido, que solo se pueden obtener consultando a través de la propiedad decorada con `@ContentChild`.

© JMA 2020. All rights reserved

AfterView

- ngAfterViewInit() y ngAfterViewChecked() se invocan después de que Angular cree las vistas secundarias de un componente.
- Actúan basándose en valores cambiantes dentro de la vista secundaria, que solo se pueden obtener consultando la vista secundaria a través de la propiedad decorada con @ViewChild.
- La regla de flujo de datos unidireccionales de Angular prohíbe las actualizaciones de la vista una vez que se haya compuesto. Ambos ganchos se disparan después de que se haya compuesto la vista del componente.
- Angular lanza un error si el gancho actualiza inmediatamente una propiedad enlazada a datos del componente, hay que posponerla a un ciclo de modificación.
`window.setTimeout(() => vm.model = value, 0);`
- Angular llama con frecuencia AfterViewChecked(), a menudo cuando no hay cambios de interés. La implementación debe ser muy ligera o la experiencia de usuario se resiente.

© JMA 2020. All rights reserved

AfterContent y AfterView

- Aunque AfterContent y AfterView son similares, la diferencia clave está en el componente hijo:
 - Los ganchos AfterContent se refieren con @ContentChildren a los componentes secundarios que Angular **proyectó** en el componente.
 - Los ganchos de AfterView se refieren con @ViewChildren a los componentes secundarios cuyas etiquetas de elemento aparecen **dentro de la plantilla** del componente.
- Otra diferencia radica en que AfterContent puede modificar inmediatamente las propiedades de enlace de datos del componente sin necesidad de esperar: Angular completa la composición del contenido proyectado antes de finalizar la composición de la vista del componente.

© JMA 2020. All rights reserved

OnDestroy

- Debe implementarse la lógica de limpieza en `ngOnDestroy()`, la lógica que debe ejecutarse antes de que Angular destruya la directiva.
- Este es el momento de notificar a otra parte de la aplicación que el componente va a desaparecer.
- Es el lugar para:
 - Liberar recursos que no serán eliminados automáticamente por el recolector de basura.
 - Darse de baja de Observables y eventos DOM.
 - Detener temporizadores.
 - Eliminar el registro de todas las devoluciones de llamada que se registraron en servicios globales o de aplicaciones.
- Nos arriesgamos a fugas de memoria y de proceso si no se hace.

© JMA 2020. All rights reserved

Detección de cambios

- El mayor reto de cualquier framework Javascript es mantener sincronizado el estado del componente con su representación en la vista mediante nodos DOM dado que el proceso de renderizado es lo más costoso a nivel de proceso.
- Angular tiene una mecanismo denominado `ChangeDetector` para detectar inconsistencias (cambios), entre el estado del componente y la vista. El desarrollador es responsable de actualizar el modelo de la aplicación y Angular, mediante la detección de cambios, es responsable de reflejar el estado del modelo en la vista.
- El `ChangeDetector` es un algoritmo ultra eficiente, optimizado para la maquina virtual de Javascript, dado que se genera (en tiempo de transpilación) un gestor de cambios específico para cada componente.
- Es un mecanismo que se comienza a ejecutar en el nodo padre (`root-element`) y que se va propagando a cada nodo hijo.
- Si se detecta un cambio no aplicado en la vista, éste generará una modificación en el árbol DOM.

© JMA 2020. All rights reserved

ChangeDetector

- En tiempo de ejecución, Angular creará clases especiales que se denominan detectores de cambios, uno por cada componente que hemos definido. Por ejemplo, Angular creará dos clases: AppComponent y AppComponent_ChangeDetector.
- El objetivo de los detectores de cambio es saber qué propiedades del modelo se utilizaron en la plantilla de un componente y que han cambiado desde la última vez que se ejecutó el proceso de detección de cambios.
- Para saberlo, Angular crea una instancia de la clase de detector de cambio apropiada con un enlace al componente que se supone que debe verificar y propiedades para almacenar los valores anteriores de las propiedades.
- Cuando el proceso de detección de cambios quiere saber si la instancia del componente ha cambiado, ejecutará el método detectChanges pasando los valores actuales del modelo para compararlos con los anteriores. Si se detecta un cambio, cachea los nuevos valores y el componente se actualiza.

© JMA 2020. All rights reserved

Como detectar cambios

- Conseguir un buen rendimiento en la aplicación, va a estar muy relacionado con cuántas veces (y con qué frecuencia), se ejecuta el proceso de detección de cambios.
- En el 99% de los casos, el cambio en el estado del componente está provocado por:
 - Eventos en la interfaz (clicks, mouseover, resizes, etc)
 - Peticiones Ajax
 - Ejecuciones dentro de temporizadores (setTimeout o setInterval)
- Se puede concluir con cierta seguridad que se deben actualizar las vistas (o comprobarlo al menos) después de producirse uno de estos tres supuestos.

© JMA 2020. All rights reserved

zone.js

- Zone.js crea un contexto de ejecución para operaciones asíncronas. Una zona se encarga de ejecutar una porción de código asíncrono, siendo capaz de conocer cuando terminan todas las operaciones asíncronas.
- Zone dispone del método run que recibe como argumento la función a ejecutar con las acciones asíncronas y termina cuando concluyen todas las acciones pendientes.
- Una zona tiene varios hooks disponibles:
 - onZoneCreated (al crear un nuevo fork)
 - beforeTask (antes de ejecutar una nueva tarea con zone.run)
 - afterTask (después de ejecutar la tarea con zone.run)
 - onError (Se ejecuta con cualquier error lanzado desde zone.run)
- Al hacer un fork en una zona se obtiene una nueva zona con todos los hooks definidos en su padre.

© JMA 2020. All rights reserved

zone.js

```
function miMetodoAsync() {
  // Peticiones AJAX
  // Operaciones setTimeout
}

const myZoneConf = {
  beforeTask: () => { console.log('Antes del run'); },
  afterTask: () => { console.log('Después del run'); }
};

const myZone = zone.fork(myZoneConf);
myZone.run(miMetodoAsync);
```

© JMA 2020. All rights reserved

NgZone

- NgZone es la implementación utilizada en Angular para la ejecución de tareas asíncronas. Es un fork (bifurcación) de zone.js con ciertas funcionalidades extra orientadas a gestionar la ejecución de componentes y servicios dentro (o fuera) de la zona de Angular.
- Será también el encargado de notificar al ChangeDetector que debe ejecutarse para buscar posibles cambios en el estado de los componentes y actualizar el DOM si fuera necesario.
- NgZone se utiliza como un servicio inyectable en componentes o servicios que expone los siguientes métodos la gestión de las zonas:
 - run: ejecuta cierto código dentro de la zona
 - runGuard: ejecuta cierto código dentro de la zona pero los errores se inyectan en onError en lugar de ser re-ejecutados
 - runOutsideAngular: ejecuta el código fuera de la zona de angular por lo que al concluir no se ejecuta ChangeDetector

© JMA 2020. All rights reserved

NgZone

- NgZone expone un conjunto de Observables que nos permiten determinar el estado actual o la estabilidad de la zona de Angular:
 - onUnstable: notifica cuando el código ha entrado y se está ejecutando dentro de la zona Angular.
 - onMicrotaskEmpty: notifica cuando no hay más microtasks en cola para su ejecución. Angular se suscribe a esto internamente para indicar que debe ejecutar la detección de cambios.
 - onStable: notifica cuándo el último onMicroTaskEmpty se ha ejecutado, lo que implica que todas las tareas se han completado y se ha producido la detección de cambios.
 - onError: notifica cuando se ha producido un error. Angular se suscribe a esto internamente para enviar errores no detectados a su propio controlador de errores, es decir, los errores que ve en su consola con el prefijo 'EXCEPCIÓN':

```
constructor(private ngZone: NgZone) {  
  this.ngZone.onStable.subscribe(() => console.log('Zone are stable'));  
  this.ngZone.onUnstable.subscribe(() => console.log('Zone are unstable'));  
  this.ngZone.onError.subscribe((error) => console.error('Error', error instanceof Error ? error.message :  
    error.toString()));  
}
```

© JMA 2020. All rights reserved

NgZone

- Cualquier código ejecutado dentro de la zona y que sea susceptible a cambios (actualización Ajax, eventos de ratón, ...) va a forzar la ejecución del detector de cambios en todo el árbol de la aplicación.
- Hay que tener en cuenta que esa detección de cambios, aunque optimizada, puede tener un coste de proceso alto, sobre todo si se ejecuta varias veces por segundo.
- En determinadas ocasiones existe un elevado número de ejecuciones asíncronas, que nos llevarían a una ejecución del detector de cambios con más frecuencia de lo deseado.
- Por ejemplo, capturar el evento mousemove en un determinado componente, invocará el ChangeDetector en todo el árbol de la aplicación con cada pixel por el que pase el puntero. Es una buena práctica capturar ese evento fuera de la zona de Angular (runOutsideAngular), aunque con cautela porque puede dar lugar a inconsistencias entre el estado de los componentes y el DOM.

© JMA 2020. All rights reserved

Estrategias de detección de cambios

- Con cada actualización en NgZone, Angular deberá ejecutar el detector de cambios en cada uno de los componentes del árbol. Para que un nodo DOM se actualice, es imprescindible que se ejecute el detector de cambios en dicho nodo, para lo cual también se debe ejecutar en todos sus antecesores hasta el document raíz.
- En principio no hay manera de asociar determinadas zonas a determinados componentes: una ejecución de cualquier método asíncrono en la zona, desencadenará la ejecución de ChangeDetector en todo el árbol de la aplicación.
- Angular ha implementado dos estrategias de detección de cambios en el estado del componente: Default y OnPush, que determinan cómo y, sobre todo, cuándo se ejecuta el ChangeDetector y en qué componentes del árbol de componentes que es la aplicación.

```
@Component({  
  changeDetection: ChangeDetectionStrategy.OnPush  
})
```

© JMA 2020. All rights reserved

ChangeDetectionStrategy.Default

- ChangeDetectionStrategy.Default es la estrategia por defecto, que hace que todo funcione correctamente sin necesidad de preocuparnos de la detección de cambios; eso sí, se realiza una especie de estrategia de fuerza bruta.
- Por cada cambio ejecutado y detectado en la zona, Angular realiza comparaciones en todas las variables referenciadas en la plantilla, tanto por referencia como por valor (incluso en objetos muy profundos), y en todos los componentes de la aplicación.
- Hay que tener en cuenta que esas comparaciones por valor, son muy costosas en cuanto a consumo de CPU, ya que deberá ir comparando cualquier objeto que esté asociado a la vista. Y es una operación que se ejecutará con cualquier cambio detectado, tenga o no que ver con el componente.
- No obstante, por norma general, en el 95% de las aplicaciones web, es una estrategia válida y que consigue un rendimiento más que aceptable.

© JMA 2020. All rights reserved

ChangeDetectionStrategy.OnPush

- En ChangeDetectionStrategy.OnPush, con cada cambio registrado en la zona, la única comprobación que se realizará por componente será la de los parámetros @Input de dicho componente.
- Únicamente se invocará el ChangeDetector de ese componente, si se detecta cambios en la referencia de los parámetros @Input del componente. La comprobación por referencia es mucho más óptima y rápida, pero puede dar lugar a ciertos problemas si no se controla bien (inmutabilidad).
- Solo cuando una referencia se actualiza será cuando se volverá a renderizar la vista. El resto de los casos que necesiten renderización requerirán un “empujón”.
- Es una estrategia mucho más barata en términos de consumo de CPU. La estrategia se hereda de contenedores a contenidos, por lo que podrá estar definida en todo el árbol de la aplicación o bien limitada a ciertas ramas.

© JMA 2020. All rights reserved

ChangeDetectorRef

- Angular ofrece el servicio denominado ChangeDetectorRef que es una referencia inyectable al ChangeDetector.
- Este servicio facilita poder gestionar el detector de cambios a voluntad, lo cual resulta muy útil cuando se utiliza la estrategia de actualización OnPush o cuando se ejecuta código fuera de la zona.
- Con ChangeDetectorRef.markForCheck() nos aseguramos que el detector de cambios del componente, y de todos sus contenedores, se ejecutará en la próxima ejecución de la zona. Una vez realizada la siguiente detección de cambios en el componente, se volverá a la estrategia OnPush.

```
myObservable.subscribe(data => {  
  // ...  
  this.changeDetectorRef.markForCheck();  
});
```

© JMA 2020. All rights reserved

ChangeDetectorRef

- ChangeDetectorRef.detach(): Para sacar el componente y todo su contenido de la futura detección de cambios de la aplicación. Las futuras sincronizaciones entre los estados del componente y las plantillas, deberá realizarse manualmente.
- ChangeDetectorRef.reattach(): Para volver a incluir el componente y todo su contenido en las futuras detecciones de cambios de la aplicación.
- ChangeDetectorRef.detectChanges(): Para ejecutar manualmente el detector de cambios en el componente y todo su contenido. Utilizado en componentes en los que se ha invocado “detach”, consiguiendo así una actualización de la vista.

© JMA 2020. All rights reserved

Optimización y rendimiento

- Una aplicación de rendimiento exigente es aquella que ejecuta un elevado número de operaciones de cambio por segundo (eventos/XHR/websocket) y con un elevado número de enlaces en la vista que deben actualizarse en base a dichas operaciones.
- Con la estrategia `ChangeDetectionStrategy.Default`, sencilla y funcional, es recomendable:
 - Ejecutar fuera de la zona cualquier evento de alta frecuencia, de manera que sus ejecuciones queden desacopladas de la ejecución de los detectores de cambios en el árbol de componentes.
 - Evitar en la medida de lo posible “getters” costosos directamente enlazados a la vista (usar memoizes o caches).
 - Desacoplar (detach) cuando sea necesario los detectores de cambios y ejecutarlos únicamente bajo demanda.

© JMA 2020. All rights reserved

Optimización y rendimiento

- La estrategia `ChangeDetectionStrategy.OnPush` supone un mayor rendimiento dado que el algoritmo de detección de cambios se va a ejecutar muchas menos veces. Pero esta estrategia va a obligar a disponer de un diseño mucho más cuidado y específico en los componentes.
- Las recomendaciones son:
 - crear (y anidar) componentes con la mayor profundidad posible. Esto implica que nuestros componentes serán más simples y reutilizables.
 - injectar propiedades en estos componentes mediante `@Input`, utilizando estrategias de inmutabilidad:
 - Utilizar `immutable.js` o `Mori`
 - Utilizar `ngrx`, que trae un estado inmutable (`redux`) a Angular
 - O directamente con `Object.assign({}, objeto)`
 - (ab)usar de los Observables que notifican cuando llega un nuevo dato: permitiendo invocar `markForCheck` en el suscriptor o delegar la suscripción directamente en la plantilla mediante `AsyncPipe`.

© JMA 2020. All rights reserved

Internacionalización I18n

- La internacionalización es el proceso de diseño y preparación de una aplicación para que pueda utilizarse en diferentes idiomas. La localización es el proceso de traducir la aplicación internacionalizada a idiomas específicos para lugares específicos.
- Angular simplifica los siguientes aspectos de la internacionalización:
 - Visualización de fechas, números, porcentajes y monedas en un formato local.
 - Preparación del texto de las plantillas de los componentes para la traducción.
 - Manejo de formas plurales de palabras.
 - Manejo de textos alternativos.
- Para la localización, se puede utilizar Angular CLI para generar la mayor parte de la plantilla necesaria para crear archivos para traductores y publicar la aplicación en varios idiomas. Una vez que haya configurado la aplicación para usar i18n, el CLI simplifica los siguientes pasos:
 - Extraer texto localizable en un archivo que puede enviar para ser traducido.
 - Construir y servir la aplicación con una configuración regional determinada, usando el texto traducido.
 - Creación de versiones en múltiples idiomas de la aplicación.
- De forma predeterminada, Angular usa la configuración regional en-US, que es el inglés tal como se habla en los Estados Unidos de América.

© JMA 2020. All rights reserved

Pipes I18n

- Los pipes DatePipe, CurrencyPipe, DecimalPipe y PercentPipe muestran los datos localizados basándose en el LOCALE_ID.
- Por defecto, Angular solo contiene datos de configuración regional para en-US. Si se establece el valor de LOCALE_ID en otra configuración regional, se debe importar los datos de configuración regional para la nueva configuración regional. El CLI importa los datos de la configuración regional cuando con ng serve y ng build se utiliza el parámetro --configuration.
ng serve --configuration=es
- Si se desea importar los datos de configuración regional de otros idiomas, se puede hacer manualmente:

```
import { registerLocaleData } from '@angular/common';
import localeEs from '@angular/common/locales/es';
import localeEsExtra from '@angular/common/locales/extra/es';
registerLocaleData(localeEs, 'es', localeEsExtra);
```

© JMA 2020. All rights reserved

Localización

- El equipo de Angular ha fijado como mejor práctica generar tantas aplicaciones diferentes como idiomas se deseen soportar, frente a una única aplicación con múltiples idiomas.
- El proceso de traducción de plantillas i18n tiene una serie de fases:
 - Marcar todos los mensajes de texto estático en las plantillas de componentes para la traducción.
 - Crear un archivo de traducción: el comando `ng xi18n` permite extraer los textos marcados en las plantillas a un archivo fuente de traducción estándar de la industria.
`ng xi18n --output-path i18n`
 - Duplicar el archivo tantas veces como idiomas se deseen utilizando como sub extensión el identificador de idioma Unicode y, opcionalmente, la configuración regional.
 - Editar el archivo de traducción duplicado correspondiente y traducir el texto extraído al idioma de destino.
 - Crear en `angular.json` las configuraciones específicas de los diferentes idiomas.
 - Fusionar el archivo de traducción completado con la aplicación:
 - `ng build --prod --configuration=es`

© JMA 2020. All rights reserved

Localización: Marcado

- El atributo `i18n` Angular marca un contenido como traducible y hay utilizarlo en cada etiqueta o atributo cuyo contenido literal deba ser traducido.
`<h1 i18n>Hello!</h1> <ng-container i18n>sin tag</ng-container>`
- `i18n` no es una directiva angular, es un atributo personalizado reconocido por las herramientas y compiladores de Angular que después de la traducción es eliminado en la compilación.
- Para traducir un mensaje de texto con precisión, el traductor puede necesitar información adicional o contexto. El traductor también puede necesitar conocer el significado o la intención del mensaje de texto dentro de este contexto de aplicación en particular.
- Como el valor del atributo `i18n` se puede agregar una descripción del mensaje de texto y, opcionalmente, precederla del sentido `<meaning>|<description>@@<id>`:
`<h1 i18n="site header|An introduction header for this sample">Hello i18n!</h1>`
- Todas las apariciones de un mensaje de texto que tengan el mismo significado tendrán la misma traducción. El mismo mensaje de texto que está asociado con significados diferentes puede tener diferentes traducciones.
- La herramienta de extracción genera una hash como identificador de cada literal, se puede especificar uno propio usando el prefijo `@@: i18n="@@myID"`

© JMA 2020. All rights reserved

Localización: Traducción

- Para realizar la traducción se añade un elemento target a continuación del elemento source, aunque el target es el único imprescindible por cada id en el fichero traducido:

```
<source>Hello!</source>
<target>¡Hola!</target>
```
- Por defecto se genera un archivo de traducción denominado messages.xlf en el Formato de archivo de intercambio de localización XML (XLIFF, versión 1.2), pero también acepta los formatos XLIFF 2 y Paquete de mensajes XML (XMB).
 - ng xi18n --i18n-format=xlf2
 - ng xi18n --i18n-format=xmb
- Los archivos XLIFF tienen la extensión .xlf. El formato XMB genera archivos de origen .xmb pero utiliza archivos de traducción .xtb
- La mayoría de las aplicaciones se traducen a más de un idioma, es una práctica estándar dedicar una carpeta a la localización y almacenar los activos relacionados, como los archivos de internacionalización, allí.

© JMA 2020. All rights reserved

Localización: Código

- No solo el texto de las plantillas requiere traducción, también todos los literales utilizados en el código.
- Angular no provee de una utilidad similar a la de las plantillas para el código pero se puede implementar el mismo mecanismo utilizado para el environment.
- Crear en la carpeta de localización un fichero denominado message.ts (por ejemplo):

```
export const messages = {
  AppComponent: {
    title: 'Word',
  }
};
```
- Crea un duplicado por idioma siguiendo el mismo convenio message.es.ts.

```
export const messages = {
  AppComponent: {
    title: 'Mundo',
  }
};
```
- Sustituir los literales del código por:

```
title = messages.AppComponent.title;
```

© JMA 2020. All rights reserved

Localización: Configuración

```
"build": {  
  "configurations": {  
    "production": { ... }  
  },  
  "es": {  
    "aot": true,  
    "outputPath": "dist/avanzado/es/",  
    "i18nFile": "src/i18n/messages.es.xlf",  
    "i18nFormat": "xlf",  
    "i18nLocale": "es",  
    "fileReplacements": [  
      {  
        "replace": "src/i18n/messages.ts",  
        "with": "src/i18n/messages.es.ts"  
      }  
    ]  
  },  
  "serve": {  
    "configurations": {  
      "production": { ... }  
    },  
    "es": {  
      "browserTarget": "avanzado:build:es"  
    }  
  }  
}
```

© JMA 2020. All rights reserved

Localización: Configuración

- Duplicar la entrada "production" dentro de "build" : "configurations" y añadir:

```
"production": { ... }  
"es": {  
  "baseHref": "/es/",  
  "outputPath": "dist/myApp/es/",  
  "i18nFile": "src/i18n/messages.es.xlf",  
  "i18nFormat": "xlf",  
  "i18nLocale": "es",  
  "fileReplacements": [  
    {  
      "replace": "src/i18n/messages.ts",  
      "with": "src/i18n/messages.es.ts"  
    }  
  ],  
  ...  
}  
}
```

- Hacer lo mismo en "serve": "configurations": {

```
"production": { ... }  
"es": {  
  "es": {  
    "browserTarget": "myApp:build:es"  
  }  
}
```

© JMA 2020. All rights reserved

Apache2 configuration

```
<VirtualHost *:80>
  ServerName www.myapp.com
  DocumentRoot /var/www
  <Directory "/var/www">
    RewriteEngine on
    RewriteBase /
    RewriteRule ^./index\.html\$ - [L]
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule (...) \$1/index.html [L]
    RewriteCond %{HTTP:Accept-Language} ^fr [NC]
    RewriteRule ^$ /fr/ [R]
    RewriteCond %{HTTP:Accept-Language} ^es [NC]
    RewriteRule ^$ /es/ [R]
    RewriteCond %{HTTP:Accept-Language} !^es [NC]
    RewriteCond %{HTTP:Accept-Language} !^fr [NC]
    RewriteRule ^$ /en/ [R]
  </Directory>
</VirtualHost>
```

© JMA 2020. All rights reserved

Módulos como bibliotecas

- Si encuentra que necesita resolver el mismo problema en más de una aplicación (o desea compartir su solución con otros desarrolladores), tiene un candidato para una biblioteca.
- Para que su solución sea reutilizable, se debe ajustar para que no dependa del código específico de la aplicación. Algunas cosas a considerar al migrar la funcionalidad de la aplicación a una biblioteca son:
 - Las declaraciones, como componentes y pipes, deben diseñarse sin estado, lo que significa que no dependen de variables externas ni las alteran. Los componentes deben exponer sus interacciones a través de entradas para proporcionar contexto y salidas para comunicar eventos a otros componentes.
 - Todas las clases o interfaces personalizadas utilizadas en componentes o servicios deben estar disponibles en la biblioteca. Cualquier elemento observable al que los componentes se suscriban internamente debe limpiarse y desecharse durante el ciclo de vida de esos componentes.
 - Si los componentes dependen de servicios, estos deben estar disponibles en la biblioteca. Los servicios deben declarar sus propios proveedores, en lugar de declarar proveedores en el NgModule o componente, permitiendo que el árbol de servicios se pueda sacudir y el compilador deje fuera del paquete el servicio si nunca se inyecta en la aplicación que importa la biblioteca. Si se registran proveedores de servicios globales o se comparten proveedores en múltiples NgModules, se deben usar los patrones de diseño forRoot() y forChild() utilizados por el RouterModule.
 - Si el código de la biblioteca o sus plantillas dependen de otras bibliotecas (como Angular Material), se debe configurar la biblioteca con esas dependencias.

© JMA 2020. All rights reserved

Bibliotecas

- Angular CLI v6 viene con soporte de biblioteca a través de ng-packagr conectado al sistema de compilación que utilizamos en Angular CLI, junto con esquemas para generar una biblioteca.
- Se puede crear una biblioteca en un espacio de trabajo existente ejecutando los siguientes comandos:
`ng generate library my-lib`
- Ahora deberías tener una biblioteca adentro `projects/my-lib`, que contiene un componente y un servicio dentro de un NgModule dentro de la carpeta `lib/src`, donde se crearan el resto de los miembros de la biblioteca.
- Puede probar y compilar la biblioteca a través de `ng test my-lib` y `ng build my-lib`.
- La nueva librería se puede utilizar directamente dentro de su mismo un espacio de trabajo dado que la generación de la biblioteca agrega automáticamente su ruta al archivo `tsconfig`.

© JMA 2020. All rights reserved

Publicación

- Aunque existen numerosas estrategias para publicar y distribuir la biblioteca de componentes, es altamente recomendado publicar la biblioteca en NPM. NPM es un registro de software en línea para compartir bibliotecas, herramientas, utilidades, paquetes, etc.
- Una vez que la biblioteca se publica en NPM, otros proyectos pueden agregar la biblioteca como una dependencia y usar los componentes dentro de sus propios proyectos.
- El archivo `package.json` es una especie de manifiesto del proyecto. Puede hacer muchas cosas: es un repositorio central de configuración de herramientas, es donde npm y yarn almacenan los nombres y versiones de todos los paquetes instalados, permite definir tareas de ejecución, ...
- El archivo puede contener información adicional para su publicación en el repositorio de NPM: `author`, `homepage`, `license`, `keywords`, `repository`, `engines`, `browserslist`, `bugs` ...

© JMA 2020. All rights reserved

Publicación de módulos con NPM

- Para poder utilizar la biblioteca en cualquier espacio de trabajo se puede publicar en el repositorio npmjs.com para que se pueda instalar vía npm install.
- Es necesario disponer de una cuenta de usuario y seguir tres pasos:
 1. Documentar en el Readme.md la biblioteca y completar en el package.json local la información complementaria como author, homepage, license, keywords, repository, ...
 2. ng build my-lib --prod
 3. cd dist/my-lib
 4. npm publish
- Como paso opcional quizás sea necesario renombrar la librería dado que el nombre debe ser único en el repositorio.
- La opción --prod se debe usar cuando se compile para publicar, ya que de antemano limpiará por completo el directorio de compilación para la biblioteca, eliminando el código anterior que queda de versiones anteriores.
- Para instalar la librería en un nuevo proyecto:
 npm install my-lib --save

© JMA 2020. All rights reserved

package.json

```
{  
  "name": "my-lib",  
  "description": "Biblioteca demo del curso",  
  "keywords": ["curso", "demos", "angular"],  
  "homepage": "https://github.com/jmagit",  
  "repository": { "type": "git", "url": "https://github.com/jmagit" },  
  "version": "0.0.1",  
  "peerDependencies": {  
    "@angular/common": "^16.0.0",  
    "@angular/core": "^16.0.0"  
  },  
  "dependencies": {  
    "tslib": "^2.3.0"  
  },  
  "sideEffects": false  
}
```

© JMA 2020. All rights reserved

Migración desde AngularJS a Angular

- La biblioteca ngUpgrade en Angular es una herramienta muy útil para actualizar cualquier cosa menos las aplicaciones muy pequeñas.
- Con él se puede mezclar y combinar AngularJS y componentes Angular en la misma aplicación y hacer que funcionen sin problemas.
- Eso significa que no se tiene que hacer el trabajo de actualización todo de una vez, ya que hay una coexistencia natural entre los dos marcos durante el período de transición.
- <https://angular.io/guide/upgrade>
- <https://vsavkin.com/migrating-angular-1-applications-to-angular-2-in-5-simple-steps-40621800a25b>

© JMA 2020. All rights reserved

Meta tags sociales

- Esto es lo que tienes que hacer para integrar en las noticias, páginas de detalle de vídeo y foto galerías de tu sitio web las etiquetas de Facebook y Twitter.
- La idea es ponérselo fácil a los sistemas de compartir en redes sociales, como Twitter, Google+, Facebook o Pinterest, para mejorar la viralidad del contenido de tu sitio web.
- Las etiquetas meta sociales son simples. Contiene los datos mínimos para poder compartir contenido en Twitter, Facebook, Google+ y Pinterest.
 - Open Graph (Facebook) <https://developers.facebook.com/docs/sharing/webmasters>
 - Twitter cards <https://developer.twitter.com/en/docs/tweets/optimize-with-cards/guides/getting-started>
- Si debemos elegir un tipo de metadatos para incluir en una página web, estos serían los datos Open Graph (Facebook). Esto se debe a que todas las plataformas pueden utilizarlo como reserva, incluyendo Twitter. Las twitter cards también mejoran la iteración de nuestros usuarios con Twitter.

© JMA 2020. All rights reserved

SEO, Google+

```
<!-- COMMON TAGS -->
<meta charset="utf-8">
<title>Titulo de la página: 60-70 characters max</title>
<!-- Search Engine -->
<meta name="description" content="Descripción larga: 150 characters for SEO, 200 characters for Twitter & Facebook">
<meta name="image" content="https://example.com/site_image.jpg">
<!-- Google Authorship and Publisher Markup -->
<link rel="author" href=" <a href="blank">https://plus.google.com/[Google+_Profile]/posts"/</a>>
<link rel="publisher" href=" <a href="blank">https://plus.google.com/[Google+_Page_Profile]"/</a>>
<!-- Schema.org markup for Google+ -->
<meta itemprop="name" content="Titulo">
<meta itemprop="description" content="Descripcion">
<meta itemprop="image" content="http://www.example.com/image.jpg">
```

© JMA 2020. All rights reserved

Twitter Card

```
<!-- Twitter Card data -->
<meta name="twitter:card" content="summary_large_image">
<meta name="twitter:site" content="@publisher_handle">
<meta name="twitter:title" content="Titulo">
<meta name="twitter:description" content="Descripcion que no supere los 200 caracteres">
<meta name="twitter:creator" content="@author_handle">
<!-- Twitter summary card with large image. Al menos estas medidas 280x150px -->
<meta name="twitter:image:src" content="http://www.example.com/image.html">
```

© JMA 2020. All rights reserved

Facebook: Open Graph

```
<!-- Open Graph data -->
<meta property="og:title" content="Titulo" />
<meta property="og:type" content="article" />
<meta property="og:url" content=" http://www.example.com/" />
<meta property="og:image" content=" http://example.com/image.jpg" />
<meta property="og:description" content="Descripcion" />
<meta property="og:site_name" content="Nombre de la web, i.e. Moz" />
<meta property="article:published_time" content="2013-09-17T05:59:00+01:00" />
<meta property="article:modified_time" content="2013-09-16T19:08:47+01:00" />
<meta property="article:section" content="Sección de la web" />
<meta property="article:tag" content="Article Tag" />
<meta property="fb:admins" content="ID de Facebook " />
```

© JMA 2020. All rights reserved

Optimización de imágenes

- La imagen que vincules en tus datos sociales, no tiene porque estar en la página, pero debería representar el contenido correctamente. Es importante utilizar imágenes de alta calidad.
- Toda plataforma social tiene distintos estándares para el tamaño de sus imágenes. Obviamente, lo más sencillo es elegir una imagen que sirva para todos los servicios:
 - Imagen de miniatura en Twitter: 120x120px
 - Imagen grande en Twitter: 280x150px
 - Facebook: los estándares varían, pero una imagen de, al menos, 200x200px, funciona mejor. Facebook recomienda imágenes grandes de hasta 1200px de ancho. En resumen, cuanto más grandes son las imágenes, más flexibilidad vas a tener.

© JMA 2020. All rights reserved

Servicios Title y Meta

- En este caso, nos gustaría establecer la etiqueta del título de la página y completar también las etiquetas meta como la descripción.
- Dado que una aplicación Angular no se puede iniciar en todo el documento HTML (etiqueta <html>), no es posible enlazar a la propiedad textContent de la etiqueta <title> ni generar con *ngFor las etiquetas <meta>, pero podemos hacer que el uso de los servicios Title y Meta.

```
constructor(private title: Title, private meta: Meta) {}  
ngOnInit() {  
  // ...  
  // SEO metadata  
  this.title.setTitle(this.model.name);  
  this.meta.addTag({name: 'description', content: this.model.description});  
  // Twitter metadata  
  this.meta.addTag({name: 'twitter:card', content: 'summary'});  
  this.meta.addTag({name: 'twitter:title', content: this.model.description});  
}
```

© JMA 2020. All rights reserved

RXJS (REACTIVE PROGRAMMING)

© JMA 2020. All rights reserved

Reactive Programming

- La Programación Reactiva (o Reactive Programming) esta orientada a programar con flujos de datos asíncronos. Estos flujos se pueden observar y reaccionar en consecuencia.
- RxJS es una biblioteca para componer programas asíncronos y basados en eventos mediante el uso de secuencias observables. Proporciona una clase principal, el Observable, clases satélite y operadores inspirados en Array#extras (map, filter, reduce, every, etc) para permitir el manejo de eventos asíncronos como colecciones.
- ReactiveX combina el patrón Observer con el patrón Iterator y la programación funcional con las colecciones para gestionar secuencias de eventos de una forma ideal.
- Los conceptos esenciales en RxJS que resuelven la administración de eventos asíncronos son:
 - Observable: representa una colección invocable de valores o eventos futuros.
 - Observer: conjunto de controladores que procesan los valores entregados por el Observable.
 - Subscription: representa la ejecución de un Observable, necesario para cancelar la ejecución.
 - Operators: son funciones puras que permiten un estilo de programación funcional de tratar las colecciones con operaciones como map, filter, concat, flatMap, etc.
 - Subject: es el equivalente a un EventEmitter y la única forma de difundir un valor o evento a múltiples observadores.
 - Schedulers: son los programadores centralizados de control de concurrencia, que permiten coordinar cuento cómputo ocurre en, por ejemplo, setTimeout, requestAnimationFrame u otros.

© JMA 2020. All rights reserved

Observables

- Los Observables proporciona soporte para pasar mensajes entre el publicador o editor y los suscriptores en la aplicación. Los observables ofrecen ventajas significativas sobre otras técnicas para el manejo de eventos, programación asíncrona y manejo de valores múltiples.
- Los observables son declarativos; es decir, definen una función para publicar valores, pero no se ejecuta hasta que un suscriptor se suscribe. El suscriptor suscrito recibe notificaciones hasta que la función se complete o hasta que cancele la suscripción.
- Un observable puede entregar múltiples valores de cualquier tipo: literales, mensajes o eventos, según el contexto.
- La API para recibir valores es la misma independientemente de que los valores se entreguen de forma síncrona o asíncrona. Debido a que la lógica de configuración y extracción es manejada por el observable, el código de la aplicación solo necesita preocuparse de suscribirse para consumir valores y, al terminar, cancelar la suscripción. Ya sea un flujo de pulsaciones de teclas, una respuesta HTTP o un temporizador de intervalos, la interfaz para escuchar los valores y detener la escucha es la misma.
- Debido a estas ventajas, Angular usa ampliamente los observables y los recomiendan para el desarrollo de aplicaciones.

© JMA 2020. All rights reserved

Anatomía de un Observable

- Los Observables se crean usando Observable.create() o un operador de creación, se suscriben con un Observer, ejecutan next/error/complete para entregar notificaciones al Observer y su ejecución puede ser eliminada.
- Estos cuatro aspectos están codificados en una instancia Observable, pero algunos de estos aspectos están relacionados con otros tipos, como Observer y Subscription.
- Las principales preocupaciones son:
 - Crear Observables
 - Suscribirse a Observables
 - Ejecutar Observables
 - Desechar Observables

© JMA 2020. All rights reserved

Anatomía de un Observable

- Los Observables se pueden crear con create, pero por lo general se utilizan los llamados operadores de creación como of, from, interval, etc.
- Suscribirse a un Observable es como llamar a una función, proporcionando devoluciones de llamadas donde se entregarán los datos.
- En una ejecución observable, se pueden entregar notificaciones de cero a infinito. Si se entrega una notificación de error o completado, entonces no se podrá entregar nada más después.
- Cuando se suscribe, obtiene una Suscripción, que representa la ejecución en curso. Basta con llamar a unsubscribe() para cancelar la ejecución.

© JMA 2020. All rights reserved

Publicador - Suscriptor

- Para crear un publicador, se crea una instancia de la clase Observable en cuyo constructor se le suministra la función de publicadora que define cómo obtener o generar valores o mensajes para publicar. Es la función que se ejecuta cuando un suscriptor llama al método subscribe().

```
let publisher = new Observable(observer => {
  // ...
  observer.next("next value");
  // ...
  return {unsubscribe() { ... }};
});
```
- Para crear un suscriptor, sobre la instancia Observable creada (publicador), se llama a su método subscribe() pasando como parámetros un objeto Observer con lo que se desea ejecutar cada vez que el publicador entregue un valor.

```
let subscriber = publisher.subscribe( { next: notify => console.log("Observer got a next value: " + notify),
  error: err => console.error("Observer got an error: " + err)
});
```
- La llamada al método subscribe() devuelve un objeto Subscription que tiene el método unsubscribe() para dejar de recibir notificaciones.

```
subscriber.unsubscribe();
```

© JMA 2020. All rights reserved

Publicador Síncrono

- En algunos casos la secuencia de valores ya está disponible pero se requiere un Observable para unificar el tratamiento de las notificaciones mediante suscriptores.
- RxJS dispone de una serie de funciones que crean una instancia observable:
 - of(...items): instancia que entrega síncronamente los valores proporcionados como argumentos.
 - from(iterable): convierte el argumento en una instancia observable. Este método se usa comúnmente para convertir una matriz en un observable.
 - empty(): instancia un observable completado.

```
const publisher = of(1, 2, 3);
let subscriber = publisher.subscribe( {
  next: notify => console.log("Observer got a next value: " + notify),
  error: err => console.error("Observer got an error: " + err)
});
```

© JMA 2020. All rights reserved

Observer

- La función de publicadora recibe un parámetro que implementa la interfaz Observer y define los métodos de devolución de llamada para manejar los tres tipos de notificaciones que un observable puede enviar:
 - **next:** obligatorio, la función invocada para cada valor a devolver. Se llama cero o más veces después de que comience la ejecución.
 - **error:** opcional, un controlador para una notificación de error. Un error detiene la ejecución de la instancia observable.
 - **complete:** opcional, controlador para notificar se ha completado la ejecución.
- Se corresponden con los controladores suministrados al crear la suscripción.

```
var observer = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};

• Alternativamente, el método subscribe() puede aceptar como argumentos los controladores next, error y complete.
```

© JMA 2020. All rights reserved

Suscripción

- Una Suscripción es un objeto que representa un recurso desecharable, generalmente la ejecución de un Observable. La Suscripción tiene el método unsubscribe, sin argumentos, para eliminar el recurso Subscription de la suscripción. En versiones anteriores de RxJS, la Subscription se llamaba "Disposable".
- Una Suscripción esencialmente solo tiene el metodo unsubscribe() para liberar recursos o cancelar ejecuciones de Observable.

```
let observable = interval(1000);
let subscription = observable.subscribe({next: x => console.log(x)});
// ...
subscription.unsubscribe();
```

- Las suscripciones también se pueden juntar, de modo que una llamada a unsubscribe() de una suscripción puede dar de baja de varias suscripciones.

```
var childSubscription = observable2.subscribe({next: x => console.log('second: ' + x)});
subscription.add(childSubscription);
subscription.unsubscribe();
```

© JMA 2020. All rights reserved

Subject

- Un Subjet RxJS es un tipo especial de Observable que permite la multidifusión de valores a varios observadores. Mientras que los Observables simples son unidifusión (cada Observador suscrito posee una ejecución independiente del Observable), los Subjet son multidifusión.
- Los Subjet son como EventEmitters: mantienen un registro de muchos oyentes.
- Cada Subjet es un observable.
 - Un Subjet permite suscripciones proporcionando un Observador, que comenzará a recibir valores normalmente.
 - Desde la perspectiva del Observer, no se puede distinguir si la ejecución proviene de un Observable unidifusión o multidifusión.
 - Internamente para el Subjet, subscribe() no invoca una nueva ejecución que entrega valores, simplemente registra el Observer dado en una lista de Observadores, de forma similar al addListener de los eventos.
- Cada Subjet es un observador.
 - Es un objeto con los métodos next(v), error(e) y complete(), la invocación ellos será difundida a todos los suscriptores del Subjet.

© JMA 2020. All rights reserved

Subject

- Como observable:

```
let subject = new Subject();
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
subject.subscribe({ next: (v) => console.log('observerB: ' + v) });
subject.next(1);
subject.next(2);
subject.complete();
```
- Como observador:

```
let subject = new Subject();
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
subject.subscribe({ next: (v) => console.log('observerB: ' + v) });
let observable = from([1, 2, 3]);
observable.subscribe(subject); //de unidifusión a multidifusión
```

© JMA 2020. All rights reserved

BehaviorSubject

- Una de las variantes de Subject es el BehaviorSubject, que tiene la noción de "valor actual". Almacena el último valor emitido a sus consumidores y, cada vez que un nuevo observador se suscriba, recibirá de inmediato el "valor actual" de BehaviorSubject.
- Los BehaviorSubjects son útiles para representar "valores a lo largo del tiempo".
- BehaviorSubject recibe en el constructor el valor inicial que el primer Observer recibe cuando se suscribe.

```
let subject = new BehaviorSubject(-1);
interval(2000).subscribe(subject);
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
setTimeout(() => subject.subscribe({
  next: (v) => console.log('observerB: ' + v)
}), 2500);
```

© JMA 2020. All rights reserved

ReplaySubject

- El ReplaySubject es similar a BehaviorSubject en que puede enviar valores antiguos a nuevos suscriptores, pero también puede registrar una parte de la ejecución Observable. Un ReplaySubject recuerda múltiples valores de la ejecución Observable y se los notifica a los nuevos suscriptores.
- Al crear un ReplaySubject se puede especificar el tamaño del búfer, cuantos de los últimos valores notificados debe recordar. Además también se puede especificar la ventana temporal en milisegundos para determinar la antigüedad de los valores grabados.
- A los nuevos suscriptores les suministrará directamente tantos valores como indique el tamaño del búfer siempre que se encuentren dentro de la ventana temporal:

```
let subject = new ReplaySubject(5, 3000);
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
let observable = interval(1000).pipe(take(10));
observable.subscribe(subject);
setTimeout(() => subject.subscribe({ next: (v) => console.log('observerB: ' + v) }), 4500);
```

© JMA 2020. All rights reserved

AsyncSubject

- AsyncSubject es una variante donde solo se envía a los observadores el último valor de la ejecución del observable y solo cuando se completa la ejecución.

```
let subject = new AsyncSubject();
subject.subscribe({ next: (v) => console.log('observerA: ' + v) });
setTimeout(() => subject.subscribe({ next: (v) =>
  console.log('observerB: ' + v) }), 1500);
let observable = interval(1000).pipe(take(3));
observable.subscribe(subject);
```

© JMA 2020. All rights reserved

ConnectableObservable

- Un Observable arranca cuando se suscribe el primer observador y empieza a notificar los valores.
- Un "Observable unidifusión simple" solo envía notificaciones a un solo Observador que recibe la serie completa de valores.
- Un "Observable multidifusión" pasa las notificaciones a través de un Subject que puede tener muchos suscriptores, que reciben los valores a partir del momento en que se suscriben.
- En determinados escenarios es necesario controlar cuando empieza a generar las notificaciones.
- Con el operador multicast, los observadores se suscriben a un Subject subyacente y el Subject se suscribe a la fuente Observable.
- El operador multicast devuelve un ConnectableObservable que es a su vez un Observable pero que dispone de los métodos connect() y refCount().

© JMA 2020. All rights reserved

connect

- El método `connect()` permite determinar cuándo comenzará la ejecución compartida del Observable y devuelve una Subscription con el `unsubscribe()` que permite cancelar la ejecución del Observable compartido.

```
let multicasted = interval(2000).pipe(multicast(() => new BehaviorSubject(0)));
let subscriberA = multicasted.subscribe({ next: (v) => console.log('observerA: ' + v) });
setTimeout(() => subject.subscribe({ next: (v) => console.log('observerB: ' + v) }), 4500);
// ...
let subscriptionConnect = multicasted.connect();
// ...
subscriberA.unsubscribe(); // Cancela la suscripción
// ...
subscriptionConnect.unsubscribe(); // Cancela la ejecución
```

© JMA 2020. All rights reserved

RefCount

- Llamar manualmente a `connect()` y manejar las suscripciones es a menudo engorroso. Por lo general, queremos conectarnos automáticamente cuando llega el primer Observer y cancelar automáticamente la ejecución compartida cuando el último Observer anula la suscripción.
- El método `refCount()` de `ConnectableObservable` devuelve un Observable que realiza un seguimiento de cuántos suscriptores tiene.
- Hace que el Observable comience a ejecutarse automáticamente cuando llega el primer suscriptor y deja de ejecutarse cuando el último suscriptor se va.

```
let multicasted = (interval(1000).pipe(multicast(() => new ReplaySubject(5, 3000))) as
  ConnectableObservable<number>).refCount();
// ...
let subscriberA = multicasted.subscribe({ next: (v) => console.log('observerA: ' + v) });
// ...
subscriberA.unsubscribe();
```

© JMA 2020. All rights reserved

Operadores

- Los operadores son funciones que se basan en un observable para permitir la manipulación sofisticada de colecciones. Por ejemplo, RxJS define operadores como map(), filter(), concat(), y flatMap().
- Los operadores reciben argumentos de configuración y devuelven una función que toma una fuente observable. Al ejecutar esta función devuelta, el operador observa los valores emitidos de la fuente observable, los transforma y devuelve un nuevo observable con los valores transformados.
- Se usan tuberías o canalizaciones para vincular varios operadores juntos. Las tuberías permiten combinar múltiples operaciones en una sola operación. El método pipe() recibe como argumentos la lista de operadores que se desea combinar, y devuelve una nueva función que, al ser ejecutada, ejecuta las funciones compuestas en secuencia.
- Un conjunto de operadores aplicados a un observable es una receta, es decir, un conjunto de instrucciones para producir los valores finales. En sí misma, la receta no hace nada. Se debe llamar a subscribe() para producir un resultado a través de la receta.
- RxJS proporciona muchos operadores (más de 150 de ellos), pero solo un puñado se usa con frecuencia.

© JMA 2020. All rights reserved

Operadores

AREA	OPERADORES
Creación	from, fromPromise, fromEvent, of, range, interval, timer, empty, throw, throwError
Combinación	combineLatest, concat, merge, startWith, withLatestFrom, zip
Filtrado	debounceTime, distinctUntilChanged, filter, take, takeUntil
Transformación	bufferTime, concatMap, map, mergeMap, scan, switchMap
Utilidad	tap, delay, toArray,toPromise
Multidifusión	share, cache, multicast, publish
Errores	catch, retry, retryWhen
Agregados	count, max, min, reduce

© JMA 2020. All rights reserved

Operadores

```
const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );
squareOdd.subscribe({ next: x => console.log(x)});
let stream$ = of(5,4,7,-1).pipe(max())
stream$ = of(5,4,7,-1).pipe(
  max((a, b) => a == b ? 0 : (a > b ? 1 : -1))
).subscribe({ next: x => console.log(x)});
const sample = val => of(val).pipe(delay(5000));
const example = sample('First Example')
  .pipetoPromise()
  .then(result => {
    console.log('From Promise:', result);
  });
});
```

© JMA 2020. All rights reserved

Scheduler

- Un scheduler controla cuándo se inicia una suscripción y cuándo se entregan las notificaciones. Son despachadores centralizados de control de concurrencia, lo que nos permite coordinar cuando sucede algún “computo”, por ejemplo setTimeout, requestAnimationFrame u otros.
- Consta de tres componentes:
 - Estructura de datos: Sabe cómo almacenar y poner en cola tareas basadas en la prioridad u otros criterios.
 - Contexto de ejecución: Denota dónde y cuándo se ejecuta la tarea. Un ejemplo sería que se ejecutara inmediatamente, también podría ejecutarse en otro mecanismo de devolución de llamada como setTimeout.
 - Reloj (virtual): Proporciona una noción de “tiempo”, para ello se usa el método now(). Las tareas que se programan en un programador en particular se adhieren sólo al tiempo indicado por ese reloj.

© JMA 2020. All rights reserved

Tipos de Scheduler

Scheduler	Propósito
null	Al no pasar ningún planificador, las notificaciones se entregan de forma síncrona y recursiva. Usado para operaciones de tiempo constante o operaciones recursivas de cola.
Rx.Scheduler.queue	Programar en una cola en el marco de evento actual (planificador de trampolín). Usado para operaciones de iteración.
Rx.Scheduler.asap	Programar en la cola de micro tareas, que utiliza el mecanismo de transporte más rápido disponible, ya sea process.nextTick() de Node.js, Web Worker MessageChannel, setTimeout u otros. Usado para conversiones asincrónicas.
Rx.Scheduler.async	La programación funciona con setInterval. Usado para operaciones basadas en tiempo.

© JMA 2020. All rights reserved

Migración v.6

- Re factorización de las rutas de importación.

```
import { Observable, Subject, asapScheduler, pipe, of, from, interval, merge, fromEvent } from 'rxjs';
import { map, filter, scan } from 'rxjs/operators';
```
- El estilo de codificación anterior de los operadores de encadenamiento se ha reemplazado por conectar el resultado de un operador a otro mediante tuberías.
- La creación de diferentes tipos de Observables mediante los correspondientes métodos de clase create() se han reemplazado por funciones y operadores.
- El cambio de estilo de codificación ha obligado a renombrar aquellos que son palabras reservadas en JavaScript:
 - do -> tap
 - catch -> catchError
 - switch -> switchAll
 - finally -> finalize

© JMA 2020. All rights reserved

Migración v.6

Antes

```
source
  .do(rsIt => console.log(rsIt))
  .map(x => x + x)
  .mergeMap(n => of(n + 1, n + 2)
    .filter(x => x % 1 == 0)
    .scan((acc, x) => acc + x, 0)
  )
  .catch(err => of('error found'))
  .subscribe(printResult);

Observable.iif(test, a$, b$);
Observable.throw(new Error());

import { merge } from 'rxjs/operators';
a$.pipe(merge(b$, c$));

obs$ = ArrayObservable.create(myArray);
```

Ahora

```
source.pipe(
  tap(rsIt => console.log(rsIt))
  map(x => x + x),
  mergeMap(n => of(n + 1, n + 2).pipe(
    filter(x => x % 1 == 0),
    scan((acc, x) => acc + x, 0),
  )),
  catchError(err => of('error found')),
).subscribe(printResult);

iif(test, a$, b$);
throwError(new Error());

import { merge } from 'rxjs';
merge(a$, b$, c$);

obs$ = from(myArray);
```

© JMA 2020. All rights reserved

Observables en Angular

- Angular hace uso de los observables como una interfaz común para manejar gran variedad de operaciones asincrónicas.
- La clase EventEmitter se extiende de Observable.
- El módulo HTTP usa observables para manejar solicitudes y respuestas AJAX.
- Los módulos de Enrutador y Formularios Reactivos usan observables para escuchar y responder a eventos de entrada de usuario.
- El AsyncPipe se suscribe a un observable o promesa y devuelve el último valor que ha emitido. Cuando se emite un nuevo valor, la tubería marca el componente a verificar para ver si hay cambios.

© JMA 2020. All rights reserved

Convenciones en la nomenclatura para observables

- Debido a que las aplicaciones Angular están escritas principalmente en TypeScript, normalmente se sabrá cuándo una variable es observable.
- Aunque el Angular no hace cumplir una convención en la nomenclatura para observables, a menudo los observables aparecen nombrados con un signo "\$" como sufijo.
- Esto puede ser útil al escanear a través del código en búsqueda de valores observables.
- Además, si desea que una propiedad almacene el valor más reciente de un observable, puede ser conveniente simplemente usar el mismo nombre con o sin el "\$".

© JMA 2020. All rights reserved

Promise Pattern

- El Promise Pattern es un patrón de organización de código que permite encadenar llamadas a métodos que se ejecutarán a la conclusión del anterior (flujos).
- Simplifica y soluciona los problemas comunes con el patrón Callback:
 - Llamadas anidadas
 - Complejidad de código
$$o.m(1, 2, f(m1(3, f1(4,5,ff(8)))) \rightarrow o.m(1, 2).f().m1(3).f1(4, 5).ff(8)$$
- Aunque se utiliza extensamente para las operaciones asíncronas, no es exclusivo de las mismas.
- Las promesas se han incorporado a los objetos estándar de JavaScript en la versión EcmaScript 2015.

© JMA 2020. All rights reserved

Objeto Promise

- Una “promesa” es un objeto que actúa como proxy en los casos en los que no se puede utilizar el verdadero valor porque aún no se conoce (no se ha generado, llegado, ...) pero se debe continuar sin esperar a que este disponible (no se puede bloquear la función esperando a su obtención).
- Una “promesa” puede tener los siguientes estados:
 - Pendiente: Aún no se sabe si se podrá o no obtener el resultado.
 - Resuelta: Se ha podido obtener el resultado (`Promise.resolve()`)
 - Rechazada: Ha habido algún tipo de error y no se ha podido obtener el resultado (`Promise.reject()`)
- Los métodos del objeto promesa devuelven al propio objeto para permitir apilar llamadas sucesivas.
- Como objeto, la promesa se puede almacenar en una variable, pasar como parámetro o devolver desde una función, lo que permite aplicar los métodos en distintos puntos del código.

© JMA 2020. All rights reserved

Crear promesas (ES2015)

- El objeto Promise gestiona la creación de la promesa y los cambios de estados de la misma.

```
list() {  
    return new Promise((resolve, reject) => {  
        this.http.get(this.baseUrl).subscribe( data => resolve(data), err => reject(err) )  
    });  
}
```
- Para crear promesas ya concluidas:
 - `Promise.reject`: Crea una promesa nueva como rechazada cuyo resultado es igual que el argumento pasado.
 - `Promise.resolve`: Crea una promesa nueva como resuelta cuyo resultado es igual que su argumento.
- Un Observable se puede convertir en una promesa:

```
import 'rxjs/add/operator/toPromise';  
list() {  
    return this.http.get(this.baseUrl).toPromise();  
}
```

© JMA 2020. All rights reserved

Invocar promesas

- El objeto Promise creado expone los métodos:
 - `then(fnResuelta, fnRechazada)`: Recibe como parámetro la función a ejecutar cuando termine la anterior y, opcionalmente, la función a ejecutar en caso de que falle la anterior.
 - `catch(fnError)`: Recibe como parámetro la función a ejecutar en caso de que falle.
`list().then(calcular, ponError).then(guardar)`
- Otras formas de crear e invocar promesas son:
 - `Promise.all`: Combina dos o más promesas y realiza la devolución solo cuando todas las promesas especificadas se completan o alguna se rechaza.
 - `Promise.race`: Crea una nueva promesa que resolverá o rechazará con el mismo valor de resultado que la primera promesa que se va resolver o rechazar entre los argumentos pasados.

© JMA 2020. All rights reserved

ANGULAR SIGNALS

© JMA 2020. All rights reserved

Reactividad de grano fino

- La reactividad de grano fino es un paradigma de programación que permite la reevaluación automática del código en respuesta a cambios en los datos o el estado. En este paradigma, el código se divide en pequeñas unidades reactivas independientes que son sensibles a los cambios en sus datos de entrada. Cuando los datos de entrada cambian, estas unidades reactivas se vuelven a evaluar automáticamente, actualizando su salida en consecuencia.
- La reactividad de grano fino puede mejorar el rendimiento y la capacidad de mantenimiento de aplicaciones complejas al reducir la necesidad de actualizaciones manuales de la interfaz de usuario y permitir un procesamiento de datos más eficiente. También puede ayudar a garantizar que la interfaz de usuario de la aplicación se mantenga coherente con los datos subyacentes en todo momento, lo que mejora la experiencia del usuario.
- La reactividad de grano fino se usa comúnmente en frameworks de programación como Solid.js, Vue.js y Svelte.

© JMA 2020. All rights reserved

Signals y ZoneJs

- Hasta ahora Angular depende de zone.js para su detección automática de cambios. ZoneJs es una librería que provee un mecanismo llamado zonas para encapsular e interceptar actividades de nuestra aplicación, es decir, cuando ocurre un cambio zonejs lo detecta y activa la detección de cambios de Angular para actualizar el estado de la aplicación, entonces, el framework (Angular) pasa por todos los componentes del árbol para verificar si su estado ha cambiado o no y si el nuevo estado afecta la vista. Si ese es el caso, se actualiza la parte DOM del componente que se ve afectada por el cambio. Esto significa que, a menudo, incluso si no tenemos la intención de actualizar el DOM, Angular deberá verificar el árbol de componentes completo y ver si alguno de los valores de nuestros enlaces de datos debe actualizarse.
- La principal diferencia entre Signals y ZoneJs es que Signals se centra en detectar cambios solo en los componentes de la interfaz de usuario que son necesarios, en lugar de realizar una detección de cambios exhaustiva en todo el árbol de componentes. Esto reduce el impacto en el rendimiento del sistema de detección de cambios y permite una actualización más rápida de la interfaz de usuario.
- Otra diferencia importante es que el uso de Signals también permite la detección de cambios asíncronos, lo que significa que los cambios realizados por eventos que ocurren fuera del ciclo de vida normal de Angular, como eventos del DOM o solicitudes HTTP, también se pueden detectar y actualizar en la interfaz de usuario de manera eficiente.

© JMA 2020. All rights reserved

Angular Signals

- Angular Signals es un sistema que realiza un seguimiento granular de cómo y dónde se usa su estado en una aplicación, lo que permite que el marco optimice las actualizaciones de representación.
- Una señal es un envoltorio alrededor de un valor que puede notificar a los consumidores interesados cuando cambia su valor. Las señales pueden contener cualquier valor, desde primitivas simples hasta estructuras de datos complejas.
- El valor de una señal siempre se lee a través de una función getter, que permite a Angular rastrear dónde se usa la señal.
- Las señales pueden ser de lectura/escritura o de solo lectura.
- Angular suministra 3 primitivas para manejar las señales:
 - signal()
 - compute()
 - effect()

© JMA 2020. All rights reserved

signal()

- La primitiva signal() crea e inicializa una señal de lectura/escritura del tipo genérico WritableSignal<T>.

```
public count: WritableSignal<number> = signal(0);
```
- Se utiliza como función (getter) para consultar o vincular su valor:

```
console.log('The count is: ' + this.count());
<output>{{count()}}</output>
```
- Para establecer directamente la señal en un nuevo valor y notificar a los dependientes:

```
– this.count.set(3);
```
- Para sustituir el valor la señal en función de su valor actual y notificar a los dependientes:

```
– this.count.update(oldValue => oldValue + 1);
```
- Cuando se trabaja con señales que contienen objetos, a veces es útil mutar ese objeto directamente sin reemplazar el objeto por completo, para ello se usa el método mutate():

```
const modes = signal([{title: 'Signals', done: false}, {title: 'Zones', done: true}]);
modes.mutate(value => { value[0].done = true; });
```
- Con el método asReadonly() se devuelve una versión de solo lectura de la señal. Se puede acceder a las señales de solo lectura para leer su valor, pero no se pueden cambiar mediante métodos set, update o mutate.

© JMA 2020. All rights reserved

compute()

- La primitiva compute() permite crear señales derivadas de otras señales. Son señales de solo lectura del tipo genérico Signal<number>.
- Para crear la señal se debe especificar la función de calculo en la que deben intervenir otras señales:

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```
- La primitiva captura las notificaciones de las señales que participan en la función de calculo, de tal forma que la función no se ejecuta para calcular su valor hasta que se lee por primera vez. Una vez calculado, el valor se almacena en caché para las futuras lecturas que devolverán el valor almacenado en caché sin volver a calcular. Cuando recibe una notificación de una de sus señales invalida la cache para que se vuelva a calcular en la próxima lectura.
- Se utiliza como función (getter) para consultar o vincular su valor.

© JMA 2020. All rights reserved

Igualdad

- Al crear una señal, se puede proporcionar una función de igualdad, que se utilizará para verificar si el nuevo valor es realmente diferente al anterior.

```
prov = signal({ id: 1, nombre: 'Madrid' }, { equal: (a, b) => a.id === b.id })
prov.set({ id: 1, nombre: 'madrid' }) // no cambia
prov.set({ id: 2, nombre: 'Madrid' }) // cambia
```
- Si la función de igualdad determina que dos valores son iguales:
 - bloquea la actualización del valor de la señal
 - omite la propagación de los cambios
- Las funciones de igualdad se pueden proporcionar tanto a las señales de lectura/escritura como a las calculadas.
- En las señales de lectura/escritura solo se utiliza en .set() y .update() porque .mutate() no verifica la igualdad porque muta el valor actual sin producir una nueva referencia.

© JMA 2020. All rights reserved

effect()

- Las señales son útiles porque pueden notificar a los consumidores interesados cuando cambian. Un efecto es una operación que se ejecuta cada vez que cambian uno o más valores de señal.
- Los efectos siempre se ejecutan al menos una vez. Cuando se ejecuta un efecto, rastrea cualquier lectura de valor de señal. Cada vez que alguno de estos valores de señal cambia, el efecto vuelve a ejecutarse. Al igual que las señales calculadas, los efectos realizan un seguimiento de sus dependencias de forma dinámica y solo rastrean las señales que se leyeron en la ejecución más reciente.
- Los efectos son raramente necesarios en la mayoría de los códigos de aplicación, pero pueden ser útiles en circunstancias específicas, podría ser una buena solución para:
 - Mostar datos de registro y cuándo cambian, ya sea para análisis o como herramienta de depuración
 - Mantener los datos sincronizados con window.localStorage
 - Agregar un comportamiento DOM personalizado que no se puede expresar con la sintaxis de la plantilla
 - Optimizar la representación personalizada en un <canvas>, una biblioteca de gráficos u otra biblioteca de interfaz de usuario de terceros
- No deben usarse los efectos para la propagación de cambios de estado porque pueden generar actualizaciones circulares infinitas o ciclos de detección de cambios innecesarios.
- Los efectos siempre se ejecutan de forma asíncrona, durante el proceso de detección de cambios.

© JMA 2020. All rights reserved

effect()

- Para registrar un nuevo efecto se requiere un "contexto de inyección" (acceso a inject()) disponibles en componentes, directivas, servicios y funciones factoría de servicios.
- Se pueden crear en el constructor:

```
constructor() {
  effect(() => { console.log(`The count is: ${this.count()}`); });
}
```
- Se pueden crear como inicializador de un atributo (campo) de la clase, que también le da un nombre descriptivo:

```
private loggingEffect = effect(() => { console.log(`The count is: ${this.count()}`); });
```
- Para crear un efecto en un método fuera del constructor, se debe pasar un Injector al effect a través de sus opciones:

```
constructor(private injector: Injector) {}

init() {
  effect(() => { console.log(`The count is: ${this.count()}`); }, {injector: this.injector});
}
```

© JMA 2020. All rights reserved

effect()

- Los efectos pueden iniciar operaciones de larga duración, que deben cancelarse si el efecto se destruye o se vuelve a ejecutar antes de que finalice la primera operación. Cuando se crea un efecto, en la función se puede definir un parámetro donde se le inyecta la función que permite registrar una devolución de llamada que se invoca antes de que comience la siguiente ejecución del efecto, o cuando se destruye el efecto.

```
effect((onCleanup) => {  
  const user = currentUser();  
  const timer = setTimeout(() => {  
    console.log(`1 second ago, the user became ${user}`);  
  }, 1000);  
  onCleanup((() => { clearTimeout(timer); }));  
});
```

© JMA 2020. All rights reserved

REpresentational State Transfer

SERVICIOS RESTFUL

© JMA 2020. All rights reserved

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, ...

© JMA 2020. All rights reserved

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)

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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/>

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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`

© JMA 2020. All rights reserved

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  
}
```

© JMA 2020. All rights reserved

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 una 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.

© JMA 2020. All rights reserved

ACCESO AL SERVIDOR

© JMA 2020. All rights reserved

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: [ HttpClientModule, HttpClientJsonpModule, // ...
  ], // ...
})
export class AppModule { }
```

© JMA 2020. All rights reserved

Características (v 4.3)

- 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

© JMA 2020. All rights reserved

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>`

© JMA 2020. All rights reserved

Opciones adicionales

- **headers:** Colección de cabeceras que acompañan la solicitud
- **params:** Colección de pares nombre/valor del QueryString
- **body:** Para peticiones POST o PUT
- **context:** Contexto compartido y mutable que pueden usar los interceptores
- **withCredentials:** true si debe enviarse las credenciales (cookies).
- **reportProgress:** true activa el seguimiento de eventos de progreso
- **responseType:** 'arraybuffer' | 'blob' | 'json' | 'text',
- **observe:** 'body' | 'events' | 'response'

© JMA 2020. All rights reserved

Peticiones al servidor

- Solicitar datos al servidor:

```
this.http.get('ws/entidad/${id} ')
  .subscribe({
    next: datos => this.listado = datos,
    error: 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({
    next: data => console.log('Success uploading', data),
    error: error => console.error('Error: ${error}')
  });
}
```

© JMA 2020. All rights reserved

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({ next: data => {
  // data is an instance of type MyModel
}});
```
- Para obtener la respuesta completa (acceso a cabeceras, status y cuerpo):

```
this.http.get<MyModel>(this.baseUrl, {observe: 'response'})
  .subscribe({ next: 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'});
```

© JMA 2020. All rights reserved

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.
- **headers:** Cabeceras de la respuesta.
- **status:** Código de estado devuelto por el servidor.
- **statusText:** Versión textual del `status`.
- **body:** Cuerpo de la respuesta en el formato establecido.

© JMA 2020. All rights reserved

Manejo de errores

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

```
this.http.get<MyModel>(this.baseUrl).subscribe({  
    next: resp => { ... },  
    error: err => { console.log('Error !!!'); }  
});
```
- Para obtener los detalles del error:

```
(err: HttpErrorResponse) => {  
    if (error.status === 0) {  
        // A client-side or network error occurred. Handle it accordingly.  
        console.error('An error occurred:', error.error);  
    } else {  
        // The backend returned an unsuccessful response code.  
        console.error(  
            `Backend returned code ${error.status}, body was: `, error.error);  
    }  
}
```

© JMA 2020. All rights reserved

Manejo de errores (v.13)

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

```
this.http.get<MyModel>(this.baseUrl)  
.pipe(catchError((err, caught) => { console.error(err); return caught; }))  
.subscribe(resp => { ... });
```
- Para obtener los detalles del error:

```
(err: HttpErrorResponse, caught: Observable<T>) => {  
    if (error.status === 0) {  
        // A client-side or network error occurred. Handle it accordingly.  
        console.error('An error occurred:', error.error);  
    } else {  
        // The backend returned an unsuccessful response code.  
        console.error(  
            `Backend returned code ${error.status}, body was: `, error.error);  
    }  
    return caught;  
}
```

© JMA 2020. All rights reserved

HttpErrorResponse

- **type:** HttpEventType.Response o HttpEventType.ResponseHeader.
- **name:** 'HttpErrorResponse'
- **ok:** Falso
- **url:** URL de la respuesta.
- **headers:** Cabeceras de la respuesta.
- **status:** Código de estado devuelto por el servidor.
- **statusText:** Versión textual del `status`.
- **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.

© JMA 2020. All rights reserved

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(...);
```

© JMA 2020. All rights reserved

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 consecutivo, 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));
```

© JMA 2020. All rights reserved

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<Array<T>> { return this.http.get<Array<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); }
}
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 qué 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>

© JMA 2020. All rights reserved

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"  
    }  
  }  
}
```

© JMA 2020. All rights reserved

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 superar 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 válido), 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>
```

© JMA 2020. All rights reserved

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&format=json`;
    return this.http.jsonp(wikiUrl, 'callback').subscribe({
      next: datos => this.items = datos,
      error: error => console.error(`Error: ${error}`)
    });
  }
}
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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({ next: 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!');
    }
  });
});
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 reintentó. Para modificarla es necesario crear una nueva con el método `clone()`:

```
return next.handle(req.clone({url: req.url.replace('http://', 'https://')}));
```
- Los interceptores deben registrarse como un servicio múltiple sobre `HTTP_INTERCEPTORS` en el orden deseado:

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

© JMA 2020. All rights reserved

Pasar metadatos a interceptores

- Las solicitudes `HttpClient` contienen un contexto que puede contener metadatos sobre la solicitud, está disponible para que los interceptores lo lean o modifiquen.
- Se almacena y recupera un valor en el contexto usando un `HttpContextToken` que establece su valor predeterminado.

```
export const RETRY_COUNT = new HttpContextToken(() => 3);
```
- Para establecer valores de contexto al realizar una solicitud:

```
this.httpClient  
.get('/data/feed', {  
    context: new HttpContext().set(RETRY_COUNT, 5),  
})
```
- Para leer valores de contexto en un interceptor:

```
const retryCount = req.context.get(RETRY_COUNT);
```

© JMA 2020. All rights reserved

Modificar la petición

```
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest, HttpContextToken } from '@angular/common/http';
export const AUTH_REQUIRED = new HttpContextToken<boolean>(() => false);
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private auth: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    if (!(req.context.get(AUTH_REQUIRED) || req.withCredentials) || !this.auth.isAuthenticated) {
      return next.handle(req);
    }
    const authReq = req.clone(
      { headers: req.headers.set('Authorization', this.auth.AuthorizationHeader) }
    );
    return next.handle(authReq);
  }
}
// http.get(url, { context: new HttpContext().set(AUTH_REQUIRED, true) })
```

© JMA 2020. All rights reserved

Modificar la respuesta

```
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest, HttpResponse } from '@angular/common/http';
@Injectable({ providedIn: 'root' })
export class AjaxWaitInterceptor implements HttpInterceptor {
  constructor(private srv: AjaxWaitService) {}
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    this.srv.Mostrar();
    return next.handle(req)
      .pipe(
        finalize(() => this.srv.Ocultar())
      );
  }
}
```

© JMA 2020. All rights reserved

Arranque independiente

- En la operación de arranque independiente, los providers de HttpClientModule se deben registrar con provideHttpClient(). Para pasar los parámetros de configuración se usan las funciones withInterceptors, withInterceptorsFromDi, withXsrfConfiguration, withNoXsrfProtection, withJsonpSupport y withRequestsMadeViaParent.
- Desde Angular v15, también se pueden usar interceptores funcionales (para la inyección de dependencia se usa inject()) del tipo HttpInterceptorFn:

```
export function ajaxWaitInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn): Observable<HttpEvent<unknown>> {
  const srv: AjaxWaitService = inject(AjaxWaitService);
  srv.Mostrar();
  return next(req).pipe(finalize(() => srv.Ocultar()));
}
```

- Los interceptores de HTTP_INTERCEPTORS se configuran con withInterceptorsFromDi() y los interceptores funcionales con withInterceptors().

```
bootstrapApplication(AppComponent, {
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true, },
    provideHttpClient(withInterceptorsFromDi(), withInterceptors([ajaxWaitInterceptor])),
  ]
});
```

© JMA 2020. All rights reserved

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.
- Angular solo lo aplica para peticiones a rutas relativas que no sean GET o HEAD.

© JMA 2020. All rights reserved

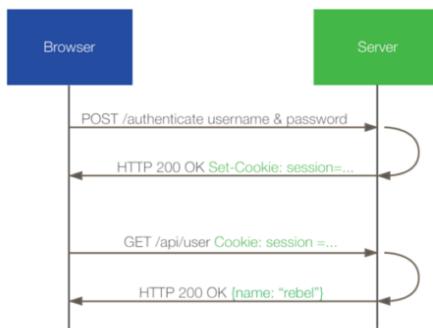
Autenticación sin estado basada en tokens

- Para solucionar los problemas de sobrecarga y escalabilidad provocados la autenticación basada en sesiones y cookies, surge la autenticación sin estado (stateless). Esto significa que el servidor no va a almacenar ninguna información, ni tampoco la sesión.
- Cuando el usuario se autentica con sus credenciales o cualquier otro método, en la respuesta recibe un token (access token) y, opcionalmente, un refresh token. El token es una cadena encriptada firmada, para evitar alteraciones y ser confiable, con una fecha de expiración corta para evitar vulnerabilidades de seguridad.
- A partir de ese momento, todas las peticiones que se hagan al API llevarán este token en una cabecera HTTP de modo que el servidor pueda identificar qué usuario hace la petición y, una vez verificado el token, confiar en las credenciales suministradas sin necesidad de buscar en base de datos ni en ningún otro sistema de almacenamiento o agente externo.
- Con este enfoque, la aplicación pasa a ser escalable, ya que es el propio cliente el que almacena su información de autenticación, y no el servidor. Así las peticiones pueden llegar a cualquier instancia del servidor y podrá ser atendida sin necesidad de sincronizaciones. Así mismo, diferentes plataformas podrán usar el mismo API. Además se incrementa la seguridad, evitando vulnerabilidades CSRF, al no existir sesiones.
- El refresh token es una identidad verificada y se usa para generar un nuevo access token cuando este expira. Típicamente, si el access token tiene fecha de expiración corta, una vez que caduca, el usuario tendrá que autenticarse de nuevo para obtener un nuevo access token. Con el refresh token, que identifica al usuario y tiene una expiración más generosa, este paso se puede saltar y con una petición al API obtener un nuevo access token que permita al usuario seguir accediendo de forma transparente a los recursos de la aplicación.

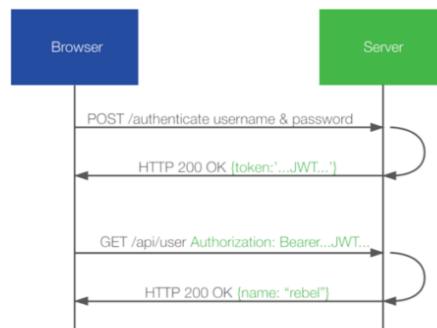
© JMA 2020. All rights reserved

Autenticación

Traditional Cookie-based Authentication



Modern Token-based Authentication



© JMA 2020. All rights reserved

OAuth 2

Flujo de protocolo abstracto



© JMA 2020. All rights reserved

Bearer Authentication

- La autenticación de portador (también llamada token de autenticación) es un [esquema de autenticación HTTP](#) que involucra tokens de seguridad llamados tokens de portador (Bearer). El nombre "Autenticación de portador" puede entenderse como "dar acceso al portador de este token". El token portador es una cadena encriptada, generalmente generada por el servidor en respuesta a una solicitud de inicio de sesión (Access token). El cliente debe enviar este token en el encabezado Authorization al realizar solicitudes a recursos protegidos:
Authorization: Bearer <token>
- El esquema de autenticación Bearer se creó originalmente como parte de OAuth 2.0 en RFC 6750, pero a veces también se usa solo. De manera similar a la autenticación básica, la autenticación de portador solo debe usarse a través de HTTPS (SSL).

© JMA 2020. All rights reserved

JWT: JSON Web Tokens

- 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 esté correctamente autenticado y saber quién 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.

<https://jwt.io>

© JMA 2020. All rights reserved

AuthService

```
@Injectable({providedIn: 'root'})
export class AuthService {
  private isAuthenticated = false;
  private authToken: string = '';
  private name = '';
  constructor() {
    if (localStorage && localStorage.AuthService) {
      const rsIt = JSON.parse(localStorage.AuthService);
      this.isAuthenticated = rsIt.isAuthenticated;
      this.authToken = rsIt.authToken;
      this.name = rsIt.name;
    }
  }
  get AuthorizationHeader() { return this.authToken; }
  get isAuthenticated() { return this.isAuthenticated; }
  get Name() { return this.name; }

  login(authToken: string, name: string ) {
    this.isAuthenticated = true;
    this.authToken = authToken;
    this.name = name;
    if (localStorage) { localStorage.AuthService =
      JSON.stringify({isAuthenticated, authToken, name}); }
  }
  logout() {
    this.isAuthenticated = false;
    this.authToken = '';
    this.name = '';
    if (localStorage) {
      localStorage.removeItem('AuthService');
    }
  }
}
```

© JMA 2020. All rights reserved

LoginService

```
@Injectable({providedIn: 'root'})
export class LoginService {
  constructor(private http: HttpClient, private auth: AuthService) { }
  get isAuthenticated() { return this.auth.isAuthenticated; }
  getName() { return this.auth.name; }
  login(usr: string, pwd: string) {
    return new Observable(observable =>
      this.http.post<LoginResponse>(environment.securityApiURL + 'login', { name: usr, password: pwd })
        .subscribe({
          next: data => {
            if (data.success === true) {
              this.auth.login(data.token ?? "", data.name ?? "");
            }
            observable.next(this.auth.isAuthenticated);
          },
          error: err => observable.error(err)
        })
    );
  }
  logout() {
    this.auth.logout();
  }
}
```

© JMA 2020. All rights reserved

ENRUTADO

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 pero requiere configurar redirecciones en el servidor para las peticiones directas.

© JMA 2020. All rights reserved

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> en index.html al principio de la etiqueta <head> para indicar al enrutador cómo componer las URL de navegación.

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

© JMA 2020. All rights reserved

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 específicas a las mas generales.
- La ruta debe obtener el componente a mostrar pero puede estar asociada a:
 - Un componente: `component: MyComponent`
 - Una tabla de sub rutas o rutas anidadas: `children: [{ path: '', component: RaizComponent }, ...]`
 - Otra ruta (redirección): `redirectTo: '/otra/ruta'`
 - Otro módulo (carga perezosa): `loadChildren: () => import('./admin/admin.module').then(mod => mod.AdminModule);`
- Adicionalmente se puede indicar:
 - `title`: Título de la página.
 - `pathMatch: prefix | full`
 - `outlet`: Destino de la ruta.
 - `data`: datos adicionales
 - Servicios guardianes (protectores de rutas): `canActivate`, `canActivateChild`, `canDeactivate`, `canLoad`

© JMA 2020. All rights reserved

Coincidencias de rutas personalizadas

- La propiedad path una poderosa estrategia de coincidencia que admite rutas estáticas, rutas variables con parámetros, rutas comodín, etc. Angular Router admite crear comparadores de rutas personalizado para situaciones en las que las URL son más complicadas.
- La función comparadora toma los siguientes argumentos:
 - segmentos: UrlSegment[]: una matriz de segmentos de URL.
 - grupo: UrlSegmentGroup: Un grupo de segmentos.
 - ruta: Route: La ruta contra la que coincidir.
- La función devuelve null si no coincide o el resultado de hacer coincidir la URL con un objeto UrlMatchResult ({ consumed: UrlSegment[]; posParams?: { [name: string]: UrlSegment; }; }).

```
export function htmlFiles(url: UrlSegment[]) {
  return url.length === 1 && url[0].path.endsWith('.html') ? ({consumed: url}) : null;
}
```
- Para utilizar un comparador personalizado se sustituye la propiedad path de la ruta por la propiedad matcher y se asocia a la función comparadora:

```
{ matcher: htmlFiles, component: AnyComponent };
```

© JMA 2020. All rights reserved

Registrar enrutamiento

```
const routes: Routes = [
  { path: '', component: HomeComponent, pathMatch: 'full', title: 'Home' },
  { 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, { bindToComponentInputs: true })
  ],
  // ...
})
export class AppModule {}
```

© JMA 2020. All rights reserved

Configuración del enrutador

- Se puede importar RouterModule varias veces, una vez por módulo. Sin embargo, solo puede estar activo un servicio Router. Para garantizar esto, hay dos formas de registrar rutas al importar el módulo:
 - El método `forRoot([])` crea un NgModule que contiene todas las directivas, las rutas dadas y el servicio Router en sí. Solo se debe usar una vez por aplicación. La opción `{ bindToComponentInputs: true }` habilita la información de enlace del estado del Router directamente a las entradas (@input) del componente
 - El método `forChild([])` crea un NgModule que contiene todas las directivas y las rutas dadas, pero no incluye al servicio Router.
- En la operación de arranque independiente, el servicio Router y las rutas dadas se configuran con `provideRouter([])`:

```
bootstrapApplication(AppComponent, {  
  providers: [ provideRouter([  
    { path: '', component: HomeComponent, pathMatch: 'full', title: 'Home' },  
    :  
  ]),  
  :  
]);
```

© JMA 2020. All rights reserved

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>`
- Cuando se aplica routerLink a un elemento en una plantilla, convierte el elemento en un enlace que inicia la navegación a una ruta (no se debe utilizar el atributo href en `<a>`):
`
<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"} ariaCurrentWhenActive="page">`

© JMA 2020. All rights reserved

Trabajar con rutas

- Los servicios Router y ActivatedRoute dan acceso a la configuración y funcionalidad de enrutador y la ruta activa.

```
import { Router, ActivatedRoute } from '@angular/router';
constructor(private router: Router, private route: ActivatedRoute) { }
```

- Para decodificar las rutas y sus parámetros:

- Usando su valor actual (Instantánea):

```
let id = this.route.snapshot.params['id'];
```

- Suscribiéndose a la 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']);
```

© JMA 2020. All rights reserved

Decodifica ruta

```
constructor(private route: ActivatedRoute) { }
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();
  }
}
```

© JMA 2020. All rights reserved

Parámetros observables

```
private obs$: any;
constructor(private router: Router, private route: ActivatedRoute) { }
ngOnInit() {
  this.obs$ = this.route.paramMap.subscribe(
    (params: ParamMap) => {
      const id = parseInt(params?.get('id') ?? '');
      if (id) {
        this.vm.edit(id);
      } else {
        this.router.navigate(['/404.html']);
      }
    });
}
ngOnDestroy() { this.obs$.unsubscribe(); }
```

© JMA 2020. All rights reserved

Vinculación como entradas de componentes

```
// imports: [RouterModule.forChild(routes, { bindToComponentInputs: true })],
// { path: 'ruta/:id', component: RutasViewComponent, data: { pageTitle: 'Contactos' } },
// http://localhost:4200/ruta/43?filtro=on
@Input() id?: string;
@Input() filtro?: string;
@Input() pageTitle?: string;
constructor(protected vm: RutasViewModelService, protected router: Router) { }
ngOnChanges(changes: SimpleChanges): void {
  if (this.id) {
    this.vm.view(this.id);
  } else {
    this.router.navigate(['/404.html']);
  }
}
```

© JMA 2020. All rights reserved

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 == '/'; }
});
```

© JMA 2020. All rights reserved

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, previen descartar acciones pendientes que se perderán.
 - CanMatch: verifica el acceso al comparador de la ruta.
 - 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 registrarla 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.isAuthenticated; }
}
{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard], },
```
- Los nuevos guardianes funcionales, permiten refactorizar este código a:

```
{ path: 'admin', component: AdminComponent, canActivate: [() => inject(AuthService).isAuthenticated], },
```

© JMA 2020. All rights reserved

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(); })
    );
  }
}
```

© JMA 2020. All rights reserved

Obtener datos antes de navegar

- En la tabla de rutas:

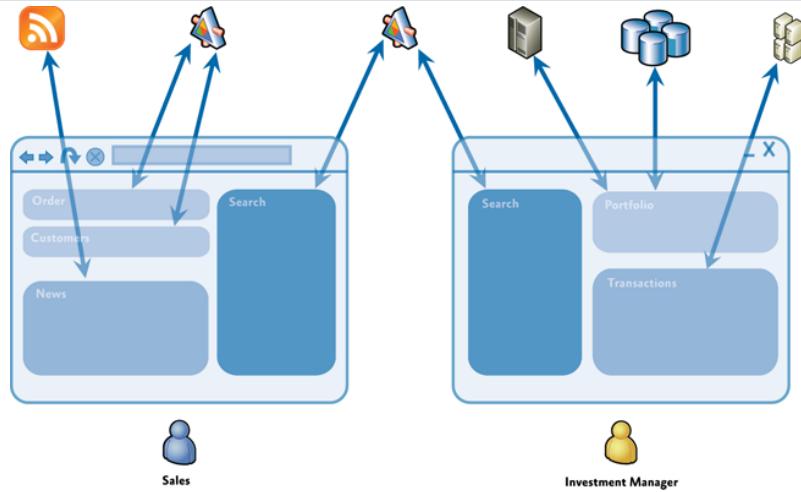
```
{ path: 'datos/:id', component: DatosViewComponent, resolve: {
  elemento: DatosResolve
},
```

- En el componente:

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

© JMA 2020. All rights reserved

Patrón Composite View



© JMA 2020. All rights reserved

outlet

- Punto de entrada con nombre (acepta binding), 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 } }}]);`

© JMA 2020. All rights reserved

Lazy loading (v8)

- A medida desarrolla la aplicación se vuelve más grande. Segundo se desarrollen nuevas áreas funcionales, el tamaño general de la aplicación seguirá creciendo. En algún momento se llegará a un punto de inflexión en el que la aplicación tarda mucho tiempo en cargarse.
- La solución es el enrutamiento asíncrono, que carga módulos de características de forma perezosa, a demanda. La carga perezosa tiene múltiples beneficios:
 - Se puede cargar áreas de funciones solo cuando lo solicite el usuario.
 - Se puede acelerar el tiempo de carga para aquellos que solo visitan ciertas áreas de la aplicación.
 - Se puede continuar expandiendo las nuevas áreas de características con carga diferida sin aumentar el tamaño del paquete de carga inicial.
 - Algunos módulos, como AppModule, deben cargarse desde el principio.
- La carga diferida de módulos (lazy loading) se controla con el router:

```
{ path: 'admin', loadChildren: () => import('./admin/admin.module')  
    .then(mod => mod.AdminModule)},
```
- Las exportaciones predeterminadas (export default) no requieren promesas:

```
{ path: 'admin', loadChildren: () => import('./admin/admin.module') },
```

© JMA 2020. All rights reserved

Lazy loading

- El módulo LazyLoad no debe estar entre los imports del @NgModule del AppModule.
- 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), // ... ] })
```
- WebPack crea un bundle por cada módulo enrutado como loadChildren identificado por un número.

© JMA 2020. All rights reserved

Lazy loading (v15)

- Cualquier ruta puede cargar un componente independiente con loadComponent:

```
{ path: 'falsa', loadComponent: () => import('./grafico-svg/grafico-svg.component'), },
```
- La operación loadChildren admite la carga de una tabla de rutas sin necesidad de escribir una carga diferida NgModule que importa RouterModule.forChild para declarar las rutas. Esto funciona cuando cada ruta cargada de esta manera usa un componente independiente.

```
{path: 'admin', loadChildren: () => import('./admin/routes').then(mod => mod.ADMIN_ROUTES)},  
// ./admin/routes.ts  
  
export const ADMIN_ROUTES: Route[] = [  
  {path: '', component: AdminHomeComponent},  
  {path: 'users', component: AdminUsersComponent},  
  // ...  
];
```
- El enrutador ahora admite especificar explícitamente providers adicional en un Route, lo que permite brindar servicios solo a un subconjunto de rutas en la aplicación.

```
{ path: 'admin', providers: [ {provide: ADMIN_API_KEY, useValue: '12345'} ], children: [ ... ]};
```

© JMA 2020. All rights reserved

API DE COMPONENTES INDEPENDIENTES

© JMA 2020. All rights reserved

Componentes independientes (v15)

- Los componentes independientes proporcionan una forma simplificada de crear aplicaciones Angular a partir de la versión 14.2 (experimental) y la 15 (estable).
- Los componentes independientes, las directivas y las canalizaciones (pipes) tienen como objetivo optimizar la experiencia de creación al reducir la necesidad de NgModules.
- Las aplicaciones existentes pueden adoptar de forma incremental y opcional el nuevo estilo independiente sin ningún cambio importante.
- Los componentes, las directivas y las canalizaciones ahora se pueden marcar como standalone: true. Las clases angulares marcadas como independientes no necesitan declararse en un NgModule (el compilador angular informará un error si lo intenta).
- Los componentes independientes especifican sus dependencias directamente en lugar de pasarla a través NgModule.
- Los imports de los componente independientes también se puede usar para hacer referencia a directivas y canalizaciones independientes. De esta forma, los componentes independientes se pueden escribir sin necesidad de crear un archivo NgModule para administrar las dependencias de la plantilla.

© JMA 2020. All rights reserved

Arranque de la aplicación

- Creación de un proyecto standalone:
 - ng new --standalone
- Modular:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```
- Operación de arranque independiente:

```
import {bootstrapApplication} from '@angular/platform-browser';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent);
```

© JMA 2020. All rights reserved

Componentes independientes (v15)

- Los componentes se pueden marcar como standalone: true y no necesitan declararse en un NgModule (da error si se intenta), pero deben especificar explícitamente sus dependencias con la propiedad imports (NgModules y componentes, directivas y pipes independientes).

```
@Component({
  selector: 'photo-gallery',
  standalone: true,
  imports: [ImageGridComponent, MatButtonModule, nglf, ngForOf],
  template: `
    <image-grid [images]="imageList"></image-grid>
    <button mat-button>Next Page</button>
  `,
})
export class PhotoGalleryComponent {
```

- Se han refactorizado las directivas y pipes integrados de los módulos estándar de Angular a sus versiones independientes para proporcionar una forma directa de importación.

© JMA 2020. All rights reserved

Inyección de Dependencias

- El módulo principal dispone de la propiedad providers para registrar los servicios e importa los providers de los módulos importado.
- La operación de arranque independiente se basa en la configuración explícita de una lista de proveedores para la inyección de dependencias. La configuración se realiza a través de la propiedad providers del segundo parámetro de bootstrapApplication().
- En Angular, las funciones de los módulos prefijadas con provide se usan para importar los providers de un módulos. Si un módulo no ofrece dichas funciones, se puede usar la utilidad importProvidersFrom() con el módulo para importar sus providers en contextos independientes.
- Para inicializar la aplicación (anteriormente en el constructor del módulo principal), se usa el nuevo token múltiple ENVIRONMENT_INITIALIZER.

```
bootstrapApplication(AppComponent, {
  providers: [
    { provide: ERROR_LEVEL, useValue: environment.ERROR_LEVEL },
    { provide: ENVIRONMENT_INITIALIZER, multi: true, useValue: () => inject(AppService).init() }
    provideRouter(MAIN_ROUTES), importProvidersFrom(SecurityModule)
  ]
});
```

© JMA 2020. All rights reserved

Interceptores

- En la operación de arranque independiente, los providers de HttpClientModule se deben registrar con provideHttpClient(). Para pasar los parámetros de configuración se usan las funciones withInterceptors, withInterceptorsFromDi, withXsrfConfiguration, withNoXsrfProtection, withJsonpSupport y withRequestsMadeViaParent.
- Desde Angular v15, también se pueden usar interceptores funcionales (para la inyección de dependencia se usa inject()) del tipo HttpInterceptorFn:

```
export function ajaxWaitInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn): Observable<HttpEvent<unknown>> {
  const srv: AjaxWaitService = inject(AjaxWaitService);
  srv.Mostrar();
  return next(req).pipe(finalize(() => srv.Ocultar()));
}
```

- Los interceptores de HTTP_INTERCEPTORS se configuran con withInterceptorsFromDi() y los interceptores funcionales con withInterceptors().

```
bootstrapApplication(AppComponent, {
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true, },
    provideHttpClient(withInterceptorsFromDi(), withInterceptors([ajaxWaitInterceptor])),
  ]
});
```

© JMA 2020. All rights reserved

Configuración del enrutador

- En la operación de arranque independiente, el servicio Router y las rutas dadas se configuran con provideRouter([]):

```
bootstrapApplication(AppComponent, {
  providers: [ provideRouter([
    { path: '', component: HomeComponent, pathMatch: 'full', title: 'Home' },
    :
  ]),
  :
}]);
```
- Cualquier ruta puede cargar un componente independiente con loadComponent:

```
{ path: 'grafico', loadComponent: () => import('./grafico-svg/grafico-svg.component'), },
{ path: 'panel', loadComponent: () => import('./admin/panel.component')
  .then(mod => mod.AdminPanelComponent)},
```

© JMA 2020. All rights reserved

Lazy loading

- La operación loadChildren admite la carga de una tabla de rutas sin necesidad de escribir una carga diferida NgModule que importa RouterModule.forChild para declarar las rutas. Esto permite muchas rutas a la vez pero solo funciona cuando cada ruta cargada de esta manera usa un componente independiente.

```
{path: 'admin', loadChildren: () => import('./admin/routes').then(mod => mod.ADMIN_ROUTES)},  
// ./admin/routes.ts  
export const ADMIN_ROUTES: Route[] = [  
  {path: '', component: AdminHomeComponent},  
  {path: 'users', component: AdminUsersComponent},  
  // ...  
];  
• Al usar loadChildren y loadComponent, el enrutador entiende y resuelve automáticamente las llamadas dinámicas import() con exportaciones default. Se puede aprovechar esto para omitir las operaciones .then() de carga diferida.  
• El enrutador ahora admite especificar explícitamente providers adicionales en un Route, lo que permite brindar servicios solo a un subconjunto de rutas en la aplicación.  
{ path: 'admin', providers: [ {provide: ADMIN_API_KEY, useValue: '12345'} ], children: [ ...
```

© JMA 2020. All rights reserved

Inyección de dependencias con rutas

- La API de carga diferida con loadChildren crea un nuevo inyector de "módulo" cuando carga los hijos de una ruta cargados de forma diferida. Esta característica a menudo era útil para brindar servicios solo a un subconjunto de rutas en la aplicación.
- El enrutador ahora admite especificar explícitamente providers adicionales en un Route, lo que permite brindar servicios solo a un subconjunto de rutas en la aplicación.
{ path: 'admin', providers: [{provide: ADMIN_API_KEY, useValue: '12345'}], children: [...
- También es posible con loadChildren combinar con providers en una configuración de enrutamiento adicional.
{path: 'admin', loadChildren: () => import('./admin/routes').then(mod => mod.ROUTES)},
export const ROUTES: Route[] = [
 { path: '', pathMatch: 'prefix', providers: [AdminService,], children: [...

© JMA 2020. All rights reserved

Migrar un proyecto existente a independiente

- A partir de la versión 15.2.0, Angular ofrece un schema para ayudar a convertir los proyectos existentes a las nuevas API independientes. El schema tiene como objetivo transformar la mayor cantidad de código posible automáticamente, pero puede requerir algunas correcciones manuales. Antes de migrar, hay que asegurarse de que el proyecto:
 - Está usando Angular 15.2.0 o posterior.
 - No tiene ningún error de compilación.
 - Está en una rama limpia de Git y todo el trabajo esta confirmado.
- Se ejecuta el esquema con el siguiente comando:
 - `ng generate @angular/core:standalone`
- El proceso de migración se compone de tres pasos por lo que se tendrá que ejecutarlo varias veces y verificar manualmente que el proyecto se compila y se comporta como se esperaba:
 1. "Convert all components, directives and pipes to standalone"
 2. "Remove unnecessary NgModule classes"
 3. "Bootstrap the project using standalone APIs"

© JMA 2020. All rights reserved

DESPLIEGUE

© JMA 2020. All rights reserved

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
 - (v 12) ng add @angular-eslint/schematics
- Para ejecutar tests unitarios: Se puede lanzar los tests unitarios con karma con el comando:
 - ng test
- Para ejecutar tests e2e: (v 12) Se puede lanzar los tests end to end con protractor con el comando: ng add @cypress/schematic
 - ng e2e

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.
 - Ahead-of-Time (AOT), que compila la aplicación antes de que la descargue el navegador (la predeterminada).
- 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, por lo que omitirlo reduce drásticamente la carga útil de la aplicación.
 - Detectar 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 que leer y ni riesgo de evaluación de HTML o JavaScript del lado del cliente, minimiza los ataques de inyección.

© JMA 2020. All rights reserved

Despliegue

- El despliegue más simple posible
 1. Generar la construcción de producción.
 2. Copiar todo dentro de la carpeta de salida (/dist por defecto o con --output-path) 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 --dev
- Paso a producción, construye optimizándolo todo para producción
 - ng build
 - 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.
- El enrulado, las aplicaciones Angular Universal y algunas funcionalidades especiales requieren una configuración especial del servidor.

© JMA 2020. All rights reserved

Distribución

📁 assets	
📄 favicon.ico	
📄 index.html	
📄 3rdpartylicenses.txt	Textos con las licencias que el empaquetado y la minimización a eliminado de los ficheros originales
📄 style.hash.css	Las definiciones de estilo de los componentes y las referenciadas en [styles] del angular.json
📄 main[-es-version].hash.js	La aplicación Angular compilada.
📄 polyfill[-es-version].hash.js	Complementos para los navegadores que no soportan todas las funcionalidades.
📄 runtime[-es-version].hash.js	Cargador de Webpack
📄 scripts[-es-versión].hash.js	Código JS de las bibliotecas de terceros referenciadas en [scripts] del angular.json
📄 lazy-module-code[-es-version].hash.js	Por cada módulo de carga diferida se crea su propio bundle.

© JMA 2020. All rights reserved

Despliegue automático (v8.3)

- El comando ng deploy ejecuta el generador de despliegue asociado al proyecto. Están disponibles varios generadores para diferentes plataformas.
- Es necesario agregar cualquiera de ellos al proyecto con ng add [package name] antes de poder usar el comando ng deploy. Cuando se agrega un paquete con capacidad de implementación, se actualizará automáticamente la sección deploy del archivo angular.json del espacio de trabajo.
 - ng add @angular/fire
 - ng deploy

Desplegar a	Paquete
Firebase hosting	@angular/fire
Azure	@azure/ng-deploy
Now	@zeit/ng-deploy
Netlify	@netlify-builder/deploy
GitHub pages	angular-cli-ghpages
NPM	ngx-deploy-npm

© JMA 2020. All rights reserved

TEST UNITARIOS

© JMA 2020. All rights reserved

Analizadores de código

- Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelven observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.
- JSHint es un analizador online de código JavaScript (basado en el JSLint creado por Douglas Crockford) que nos permitirá mostrar puntos en los que tu código no cumpla unas determinadas reglas establecidas de “código limpio”.
- El funcionamiento de JSHint es el siguiente: toma nuestro código, lo escanea y, si encuentra un problema, devuelve un mensaje describiéndolo y mostrando su ubicación aproximada.
- Para descargar e instalar:
 - `npm install -g jshint`
- Existen “plug-in” para la mayoría de los entornos de desarrollo (<http://jshint.com>). Se puede automatizar con GRUNT o GULP.

© JMA 2020. All rights reserved

ESLint

- ESLint (<https://eslint.org/>) es una herramienta para identificar e informar sobre patrones encontrados en código ECMAScript/JavaScript, con el objetivo de hacer que el código sea más consistente y evitar errores.
- Se puede instalar ESLint usando npm:
 - `npm install eslint --save-dev`
- Luego se debe crear un archivo de configuración `.eslintrc.json` en el directorio, se puede crear con `--init`:
 - `npx eslint --init`
- Se puede ejecutar ESLint con cualquier archivo o directorio:
 - `npx eslint **/*.js`
- Para habilitarlo y ejecutarlo en las últimas versiones de Angular:
 - `ng add @angular-eslint/schematics`
 - `ng lint`

© JMA 2020. All rights reserved

Jasmine

- Jasmine (<https://jasmine.github.io/>) es un framework de desarrollo dirigido por comportamiento (behavior-driven development, BDD) para probar código JavaScript.
 - No depende de ninguna otra librería JavaScript.
 - No requiere un DOM.
 - Tiene una sintaxis obvia y limpia para que se pueda escribir pruebas fácilmente.
- En resumen, podríamos decir que desde que los creadores del conocido PivotalTracker sacaron a la luz este framework de test, prácticamente se ha convertido en el estándar de facto para el desarrollo con JavaScript.
- Para su instalación “standalone”, descargar y descomprimir:
 - <https://github.com/jasmine/jasmine/releases>
- Mediante npm: (node)
 - `npm install -g jasmine`

© JMA 2020. All rights reserved

SpecRunner.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Jasmine Spec Runner v2.5.0</title>
<link rel="shortcut icon" type="image/png" href="lib/jasmine-2.5.0/jasmine_favicon.png">
<link rel="stylesheet" href="lib/jasmine-2.5.0/jasmine.css">
<script src="lib/jasmine-2.5.0/jasmine.js"></script>
<script src="lib/jasmine-2.5.0/jasmine-html.js"></script>
<script src="lib/jasmine-2.5.0/boot.js"></script>
<script type="text/javascript" src="angular.js"></script>
<script type="text/javascript" src="angular-mocks.js"></script>
<!-- include source files here... -->
<script src="src/Player.js"></script>
<script src="src/Song.js"></script>
<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
</head>
<body></body>
</html>
```

© JMA 2020. All rights reserved

Test Runner de Navegador

- Instalar el paquete como dependencia de desarrollo:
 - npm install --save-dev jasmine-browser-runner
- Inicializar para crear directorios y configuración:
 - npx jasmine-browser-runner init
- Ajustar configuración en:
 - spec/support/jasmine-browser.json
- Incorporar el comando al package.json
 - "scripts": { "test": "jasmine-browser-runner runSpecs" }
- Para ejecutar los test interactivamente:
 - npx jasmine-browser-runner serve
- Ejecutar los test:
 - npm test

© JMA 2020. All rights reserved

ESLint for Jasmine

- Instalar el paquete como dependencia de desarrollo:
 - npm install --save-dev eslint-plugin-jasmine
- Ajustar configuración en .eslintrc:

```
plugins:  
  - jasmine  
  
env:  
  jasmine: true  
  
extends: 'plugin:jasmine/recommended'
```

© JMA 2020. All rights reserved

Suites

- Una “suite” es un nombre que describe a qué género o sección se va a pasar por un conjunto de pruebas unitarias, además es una herramienta que es el núcleo que se necesita para poder tener un orden al momento de crear las pruebas.
- Las “suites” se crean con la función **describe**, que es una función global y con la cual se inicia toda prueba unitaria, además consta con dos parámetros:

```
describe("Una suite es sólo una función", function() {  
  //...  
});
```
- El primer parámetro es una cadena de caracteres donde se define el nombre de la prueba unitaria.
- El segundo parámetro es una función donde está el código que ejecutará con la prueba de código.
- Se pueden anidar “suites” para estructurar conjuntos complejos y facilitar la legibilidad y la búsqueda, basta con crear un describe dentro de otro.

© JMA 2020. All rights reserved

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Jasmine es una afirmación que debe ser verdadera pero puede ser falsa.
- Una especificación con todas las expectativas verdaderas es una especificación que pasa la prueba, pero con una o más falsas es una especificación que falla.
- Las especificaciones se definen dentro de una Suite llamando a la función global del Jasmine **it**, que al igual que describe, recibe una cadena y una función, mas un timeout opcional a cuya expiración fallara. La cadena es el título de la especificación y la función es la especificación o prueba.

```
it("y así es una especificación", function() {  
  //...  
});
```
- **describe** y **it** son funciones: pueden contener cualquier código ejecutable necesario para implementar la prueba y se aplican las reglas de alcance de JavaScript: las variables declaradas en un describe están disponibles para cualquier bloque it dentro de la suite.

© JMA 2020. All rights reserved

Expectativas

- Las expectativas se construyen con la función expect que obtiene un valor real de una expresión y lo comparan mediante una función comparadora (matcher) con un el valor esperado (constante).

```
expect(valor actual).matchers(valor esperado);
```
- Los matchers son funciones que implementan comparaciones booleanas entre el valor actual y el esperado, ellos son los responsables de informar a Jasmine si la expectativa se cumple o es falsa.
- Cualquier comparador puede evaluarse como una afirmación negativa mediante el encadenamiento a la llamada expect de un not antes de llamar al comparador.

```
expect(valor actual).not.matchers(valor esperado);
```
- También existe la posibilidad de escribir comparadores personalizados para cuando el dominio de un proyecto consiste en afirmaciones específicas que no se incluyen en los ya definidos.
- Cuando hay múltiples expectativas se puede documentar la expectativa concreta que falla.

```
expect(valor actual).withContext('mensaje a mostrar').matchers(valor esperado);
```

© JMA 2020. All rights reserved

Comparadores

- .toEqual(y); verifica si ambos valores son iguales ==.
- .toBe(y); verifica si ambos objetos son idénticos ===.
- .toMatch(pattern); verifica si el valor pertenece al patrón establecido.
- .toBeDefined(); verifica si el valor está definido.
- .toBeUndefined(); verifica si el valor es indefinido.
- .toBeNull(); verifica si el valor es nulo.
- .toBeNaN(); verifica si el valor es NaN.
- .nothing(); no espera nada, nunca falla.
- .toBeCloseTo (n, d); verifica la precisión matemática (número de decimales).
- .toContain(y); verifica si el valor actual contiene el esperado.
- .toBeInstanceOf(tipo); verifica si es del tipo esperado.
- .toHaveClass(className); verifica que la etiqueta tenga el class esperado.
- .toHaveSize(y); verifica que la colección tenga el tamaño esperado.

© JMA 2020. All rights reserved

Comparadores

- .toBeTruthy(); verifica si el valor es verdadero.
- .toBeTrue(); verifica si el valor es estrictamente true.
- .toBeFalsy(); verifica si el valor es falso.
- .toBeFalsy(); verifica si el valor es estrictamente false.
- .toBeNegativeInfinity(); verifica si el valor es infinito negativo.
- .toBePositiveInfinity(); verifica si el valor es infinito positivo.
- .toBeLessThan(y); verifica si el valor actual es menor que el esperado.
- .toBeLessThanOrEqual (y); verifica si el valor actual es menor o igual que el esperado.
- .toBeGreaterThanOrEqual (y); verifica si el valor actual es mayor o igual que el esperado.
- .toBeGreaterThanOrEqual (y); verifica si el valor actual es mayor que el esperado.
- .toThrow(); verifica si una función lanza una excepción.
- .toThrowError(e); verifica si una función lanza una excepción específica.

© JMA 2020. All rights reserved

Forzar fallos

- La función fail(msg) hace que una especificación falle. Puede llevar un mensaje de fallo o error de un objeto como un parámetro.

```
describe("Una especificación utilizando la función a prueba", function() {  
  var foo = function(x, callBack) {  
    if (x) {  
      callBack();  
    }  
  };  
  it("no debe llamar a la devolución de llamada", function() {  
    foo(false, function() {  
      fail("Devolución de llamada ha sido llamada");  
    });  
  });  
});
```

© JMA 2020. All rights reserved

Montaje y desmontaje

- Para montar el escenario de pruebas suele ser necesario definir e inicializar un conjunto de variables. Para evitar la duplicidad de código y mantener las variables inicializadas en un solo lugar además de mantener la modularidad, Jasmine suministra las funciones globales :

- beforeEach(fn) se ejecuta antes de cada especificación dentro del “describe”.
- beforeAll(fn) se ejecuta solo una vez antes empezar a ejecutar las especificaciones del “describe”.
- afterEach(fn) se ejecuta después de cada especificación dentro del “describe”.
- afterAll(fn) se ejecuta solo una vez después de ejecutar todas las especificaciones del “describe”.

```
describe("operaciones aritméticas", function(){  
  var cal;  
  beforeEach(function(){  
    calc = new Calculadora();  
  });  
  it("adicion", function(){  
    expect(calc.suma(4)).toEqual(4);  
  });  
  it("multiplicacion", function(){  
    expect(calc.multiplica(7)).toEqual(0);  
  });  
  // ...
```

- Otra manera de compartir las variables entre una beforeEach, it y afterEach es a través de la palabra clave this. Cada expectativa beforeEach/it/afterEach tiene el mismo objeto vacío this que se restablece de nuevo a vacío para la siguiente expectativa beforeEach/it/afterEach.

© JMA 2020. All rights reserved

Desactivación parcial

- Las Suites se pueden desactivar renombrando la función describe por xdescribe. Estas suites y las especificaciones dentro de ellas se omiten cuando se ejecuta y por lo tanto sus resultados no aparecerán entre los resultados de la prueba.
- De igual forma, las especificaciones se desactivan renombrando it por xit, pero en este caso aparecen en los resultados como pendientes (pending).
- Cualquier especificación declarada sin un cuerpo función también estará marcada pendiente en los resultados.

```
it('puede ser declarada con "it", pero sin una función');
```
- Y si se llama a la función de pending en cualquier parte del cuerpo de las especificaciones, independientemente de las expectativas, la especificación quedará marcada como pendiente. La cadena que se pasa a pending será tratada como una razón y aparece cuando termine la suite.

```
it('se puede llamar a "pending"en el cuerpo de las especificaciones', function() {  
    expect(true).toBe(false);  
    pending('esto es por lo que está pendiente');  
});
```

© JMA 2020. All rights reserved

Ejecución de pruebas específicas

- En determinados casos (desarrollo) interesa limitar las pruebas que se ejecutan. Si se pone el foco en determinadas suites o especificaciones solo se ejecutarán las pruebas que tengan el foco, marcando el resto como pendientes.
- Las Suites se enfocan renombrando la función describe por fdescribe. Estas suites y las especificaciones dentro de ellas son las que se ejecutan.
- De igual forma, las especificaciones se enfocan renombrando it por fit.
- Si se enfoca una suite que no tiene enfocada ninguna especificación se ejecutan todas sus especificaciones, pero si tiene alguna enfocada solo se ejecutarán las que tengan el foco.
- Si se enfoca una especificación se ejecutara independientemente de que su suite esté o no enfocada.
- La funciones de montaje y desmontaje se ejecutarán si la suite tiene alguna especificación con foco.
- Si ninguna suite o especificación tiene el foco se ejecutarán todas las pruebas normalmente.

© JMA 2020. All rights reserved

Pruebas asíncronas

- El código asíncrono es común y Jasmine necesita saber cuándo finaliza el trabajo asíncrono para evaluar el resultado.
- Jasmine admite dos formas de gestionar el trabajo asíncrono: basado en promesas (async/ await) o en devoluciones de llamada.

```
it('does a thing', async function() {
  const result = await someAsyncFunction();
  expect(result).toEqual(someExpectedValue);
});
```
- Cuando no se puedan utilizar promesas, si la función de la especificación, montaje o desmontaje define un argumento (tradicionalmente llamado done), Jasmine pasará una función para ser invocada cuando se haya completado el trabajo asíncrono, debe ser lo último que realice la función asíncrona o cualquiera de las funciones a las que llama para evitar errores o anomalías.

```
it('does a thing', function(done) {
  someAsyncFunction(function(result) {
    expect(result).toEqual(someExpectedValue);
    done();
  });
});
```

© JMA 2020. All rights reserved

Comparadores asíncronos

- Los comparadores asíncronos operan con un valor real que es una promesa y devuelven una promesa.
- La mayoría de los comparadores asíncronos esperarán indefinidamente a que la promesa se resuelva o se rechace, lo que dará como resultado la especificación falle por timeout si nunca sucede.
- Son accesibles mediante `expectAsync(promesa)` o `expectAsync(promesa).already` (cuando no está pendiente):

```
await expectAsync(promesa).toBeResolved().then(function() {
  // more spec code
});
await expectAsync(aPromise).toBeRejected();
```
- Los comparadores asíncronos son: `toBePending()`, `toBeResolved()`, `toBeResolvedTo(expected)`, `toBeRejected()`, `toBeRejectedWith(expected)`, `toBeRejectedWithError(expected, message)`.
- Como cualquier comparador, pueden evaluarse como una afirmación negativa mediante el encadenamiento de un `not` antes de llamar al comparador.

© JMA 2020. All rights reserved

Pruebas dinámicas

- Se pueden generar pruebas dinámicamente utilizando JavaScript para crear múltiples funciones de especificación.

```
describe('Calculos', () => {
  describe('Sumas', function () {
    [[1, 2, 3], [2, 2, 4], [3, -2, 1]].forEach(item => {
      it(`Prueba que ${item[0]} mas ${item[1]} es ${item[2]}`, () =>
        expect(item[0] + item[1]).toBe(item[2]))
    });
  });
})
```

© JMA 2020. All rights reserved

Dependencias

- Las dependencias con sistemas externos afectan a la complejidad de la estrategia de pruebas, ya que es necesario contar con sustitutos de estos servicios externos durante el desarrollo. Ejemplos típicos de estas dependencias son servicios web, sistemas de envío de correo, fuentes de datos o simplemente dispositivos hardware.
- Estos sustitutos, dobles de pruebas, muchas veces son exactamente iguales que el servicio original, pero en otro entorno, o son simuladores, que exponen el mismo interfaz pero realmente no realizan las mismas tareas que el sistema real, o las realizan contra un entorno controlado.
- Para poder emplear la técnica de simulación de objetos se debe diseñar el código a probar de forma que sea posible trabajar con los objetos reales o con los objetos simulados:
 - Doble herencia
 - IoC: Inversión de Control (Inversion Of Control)
 - DI: Inyección de Dependencias (Dependency Injection)
 - Simuladores de objetos

© JMA 2020. All rights reserved

Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Siguiendo la misma regla de oro, las pruebas de integración y sistema deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- Usar dobles de prueba (como los dobles en el cine) tiene ventajas:
 - Devuelven resultados determinísticos
 - Permiten crear o reproducir determinados estados
 - Obtiene resultados mucho más rápidamente y a menor coste, incluso offline.
 - Permiten el inicio temprano de las pruebas incluso cuando las dependencias todavía no están disponibles.
 - Permiten incluir atributos o métodos exclusivamente para el testeo.
 - Memorizan los valores con los que se llama a cada uno de sus miembros
 - Permiten verificar si los valores esperados coinciden con los recibidos

© JMA 2020. All rights reserved

Simulación de objetos

- **Fixture:** Es el término se utiliza para hablar de los datos de contexto de las pruebas, aquellos que se necesitan para construir el escenario que requiere la prueba.
- **Dummy:** Objeto que se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para llenar listas de parámetros.
- **Fake:** Objeto que tiene una implementación que realmente funciona pero, por lo general, usa una simplificación que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub:** Objeto que proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas (**spy**).
- **Mock:** Objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles.

© JMA 2020. All rights reserved

Espías

- Jasmine tiene funciones dobles de prueba llamados espías.
- Un espía puede interceptar cualquier función y hacer un seguimiento a las llamadas y todos los argumentos.

```
beforeEach(function() {
  fnc = spyOn(calc, 'suma');
  prop = spyOnProperty(calc, 'pantalla', 'set')
});
```
- Un espía sólo existe en el bloque describe o it en que se define, y se eliminará después de cada especificación.
- Hay comparadores (matchers) especiales para interactuar con los espías.
 - .toHaveBeenCalled(): pasará si el espía fue llamado.
 - .toHaveBeenCalledTimes(n) pasará si el espía se llama el número de veces especificado.
 - .toHaveBeenCalledWith(...) pasará si la lista de argumentos coincide con alguna de las llamadas grabadas a la espía.
 - .toHaveBeenCalledBefore(esperado): pasará si el espía se llama antes que el espía pasado por parámetro.

© JMA 2020. All rights reserved

Seguimiento de llamadas

- El proxy del espía añade la propiedad calls que permite:
 - all(): Obtener la matriz de llamadas sin procesar para este espía.
 - allArgs(): Obtener todos los argumentos para cada invocación de este espía en el orden en que fueron recibidos.
 - any(): Comprobar si se ha invocado este espía.
 - argsFor(indice): Obtener los argumentos que se pasaron a una invocación específica de este espía.
 - count(): Obtener el número de invocaciones de este espía.
 - first(): Obtener la primera invocación de este espía.
 - mostRecent(): Obtener la invocación más reciente de este espía.
 - reset(): Restablecer el espía como si nunca se hubiera llamado.
 - saveArgumentsByValue(): Establecer que se haga un clon superficial de argumentos pasados a cada invocación.

```
spyOn(foo, 'setBar');
expect(foo.setBar.calls.any()).toEqual(false);
foo.setBar();
expect(foo.setBar.calls.count()).toBe(1);
```

© JMA 2020. All rights reserved

Cambiar comportamiento

- Adicionalmente el proxy del espía puede añadir los siguientes comportamientos:
 - callFake(fn): Llamar a una implementación falsa cuando se invoca.
 - callThrough(): Llamar a la implementación real cuando se invoca.
 - exec(): Ejecutar la estrategia de espionaje actual.
 - identity(): Devolver la información de identificación para el espía.
 - returnValue(valor): Devolver un valor cuando se invoca.
 - returnValues(... values): Devolver uno de los valores especificados (secuencialmente) cada vez que se invoca el espía.
 - stub(): No haga nada cuando se invoca. Este es el valor predeterminado.
 - throwError(algo): Lanzar un error cuando se invoca.

```
spyOn(foo, "getBar").and.returnValue(745);
spyOn(foo, "getBar").and.callFake(function(arguments, can, be, received) {
  return 745;
});
spyOn(foo, "forbidden").and.throwError("quux");
```

© JMA 2020. All rights reserved

Reloj simulado

- Hay situaciones en las que es útil controlar el objeto Date y los temporizadores para anular su comportamiento o evitar pruebas lentas:
 - Operaciones que dependen de marcas temporales obtenida a través del objeto Date.
 - Operaciones diferidas con setTimeout donde no es necesario esperar.
 - Sondeos con setInterval que se quieren acelerar.
- El reloj simulado de Jasmine se utiliza al probar el código dependiente del tiempo.
- jasmine.clock().install() anula las funciones globales nativas relacionadas con el tiempo para que puedan ser controladas sincrónicamente a través de jasmine.clock().tick(), que mueve el reloj hacia adelante, ejecutando los tiempos de espera en cola por el camino. Esto incluye controlar: setTimeout, clearTimeout, setInterval, clearInterval y el objeto Date.
- Por defecto el reloj comienza en la época de Unix (marca de tiempo de 0). Esto significa que cuando se cree una instancia new Date en la aplicación, es inicializada al 01/01/1970 00:00:00. La marca temporal inicial se puede modificar con jasmine.clock().mockDate() .

© JMA 2020. All rights reserved

Reloj simulado

- `jasmine.clock().uninstall()` se restauran los métodos integrados originales.
`jasmine.clock().withMock()` permite ejecutar una función con un reloj simulado: llamará a `install` antes de ejecutar la función y a `uninstall` después de que se complete la función.

```
beforeEach(function() {
  jasmine.clock().install();
});
afterEach(function() { jasmine.clock().uninstall(); });
it('does something after 10 seconds', function() {
  const callback = jasmine.createSpy('callback');
  doSomethingLater(callback);
  jasmine.clock().tick(10000);
  expect(callback).toHaveBeenCalledWith(12345);
});
```

© JMA 2020. All rights reserved

Solicitudes de red

- Las solicitudes a recursos externos tienen el potencial de impactar negativamente en las ejecuciones de prueba debido a tiempos de carga lentos.
- En otros escenarios es difícil de obtener estados específicos del servidor, incluyendo `status`, `headers` o `body` de la respuesta o retrasos de la red, para poder realizar los casos de pruebas apropiados.
- Jasmine suministra un complemento llamado `jasmine-ajax` que permite simular las llamadas ajax en las pruebas. Para usarlo, se debe descargar el archivo `mock-ajax.js` y agregarlo a los ayudantes de Jasmine para que se cargue antes de cualquier especificación que lo use (<https://github.com/jasmine/jasmine-ajax>).
- `jasmine-ajax` proporciona un sustituto del objeto `XMLHttpRequest` que intercepta las peticiones y simula la respuesta de la solicitud.
- Las respuestas simuladas pueden establecer directamente estados específicos.
- Las simulaciones son extremadamente rápidas, la mayoría de las respuestas se devolverán en menos de 20 ms. Dado que las respuestas reales pasan por cada capa del servidor (controladores, modelos, vistas, etc.) y, es posible, que se deba inicializar una fuente de datos antes de cada prueba para generar un estado predecible, pueden ser mucho más lentas que las respuestas simuladas.

© JMA 2020. All rights reserved

Solicitudes de red

```
describe('AJAX Mock', () => {
  beforeEach(function () { jasmine.Ajax.install(); });
  afterEach(function () { jasmine.Ajax.uninstall(); });

  it("specifying response when you need it", function () {
    var doneFn = jasmine.createSpy("success");
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function (args) {
      if (this.readyState == this.DONE) {
        doneFn(this.responseText);
      }
    };
    xhr.open("GET", "/some/cool/url");
    xhr.send();
    expect(jasmine.Ajax.requests.mostRecent().url).toBe('/some/cool/url');
    expect(doneFn).not.toHaveBeenCalled();
    jasmine.Ajax.requests.mostRecent().respondWith({
      "status": 200, "contentType": 'text/plain', "responseText": 'awesome response'
    });
    expect(doneFn).toHaveBeenCalledWith('awesome response');
  });
});
```

© JMA 2020. All rights reserved

Depuración de pruebas

1. Seleccionar la ventana del navegador Karma.
2. Hacer clic en el botón DEBUG; se abre una nueva pestaña del navegador que permite volver a ejecutar las pruebas.
3. Abrir “Herramientas de Desarrollo” del navegador.
4. Seleccionar la sección de código fuentes.
5. Abrir el archivo con el código de prueba.
6. Establecer un punto de interrupción en la prueba.
7. Actualizar el navegador, que se detiene en el punto de interrupción.

© JMA 2020. All rights reserved

Pruebas Angular

- Angular, por defecto, se encuentra alineado con la calidad de software.
- Cuando Angular-CLI crea un nuevo proyecto:
 - Descarga [TSLint](#), Jazmine, Karma y [Protractor \(e2e\)](#)
 - Configura el entorno de pruebas
 - Habilita un servidor Karma de pruebas continuas (puerto: 9876)
- Así mismo, cuando genera un nuevo elemento, crea el correspondiente fichero de pruebas, usando .spec como sub extensión.
- Para ejecutar el servidor de pruebas:
 - **ng test --code-coverage** (alias: -cc)
 - **ng test --single-run** (alias: -sr)
- No se debe cerrar la instancia de Chrome mientras duren las pruebas.
- Angular suministra una serie de clases, funciones, mock y módulos específicos para las pruebas, comúnmente denominadas Utilidades Angular para pruebas.
- Permite la creación tanto de pruebas unitarias aisladas como casos de prueba que interactúan dentro del entorno Angular.

© JMA 2020. All rights reserved

Pruebas Unitarias Aisladas

- Las Pruebas Unitarias Aisladas examinan una instancia de una clase por sí misma sin ninguna dependencia Angular o de los valores inyectados.
- Se crea una instancia de prueba de la clase con new, se le suministran los parámetros al constructor y se prueba la superficie de la instancia.
- Se pueden escribir pruebas unitarias aisladas para pipes y servicios.
- Aunque también se puede probar los componentes de forma aislada, las pruebas aisladas no revelan cómo interactúan entre si los elementos Angular. En particular, no pueden revelar cómo una clase de componente interactúa con su propia plantilla o con otros componentes.

© JMA 2020. All rights reserved

Pruebas Unitarias Aisladas

- De servicios sin dependencias
 - let srv = new MyService();
- De servicios con dependencias
 - let srv = new MyService(new OtherService());
- De Pipes
 - let pipe = new MyPipe();
 - expect(pipe.transform('abc')).toBe('Abc');
- De la clase del componente:
 - let comp = new MyComponent();
- De la clase del componente con dependencias:
 - let comp = new MyComponent(new MyService());

© JMA 2020. All rights reserved

Utilidades Angular para pruebas

- Para realizar pruebas dentro del contexto de Angular, las Utilidades Angular para pruebas cuentan con las clases TestBed, ComponentFixture, DebugElement y By, así como varias funciones de ayuda para sincronizar, injectar, temporizar, ...
- Para probar los componentes, lo mas correcto es crear dos juegos de pruebas, a menudo en el mismo archivo de especificaciones.
 - Un primer conjunto de pruebas aisladas que examinan la corrección de la clase del componente.
 - Un segundo conjunto de pruebas que examina como se comporta el componente dentro del Angular, como interactúa con las plantillas, si actualiza el DOM y colabora con el resto de la aplicación.

© JMA 2020. All rights reserved

TestBed

- TestBed representa un módulo Angular para la prueba, proporciona el medio ambiente del módulo para la clase que desea probar. Extrae el componente a probar desde su propio módulo de aplicación y lo vuelve a conectar al módulo de prueba construido a medida, de forma dinámica, específicamente para una serie de pruebas.
- El método configureTestingModule reemplaza a la anotación @NgModule en la declaración del módulo. Recibe un objeto @NgModule que puede tener la mayoría de las propiedades de metadatos de un módulo normal de Angular.
- El estado base incluye la configuración del módulo de prueba predeterminado con las declaraciones (componentes, directivas y pipes) y los proveedores (servicios inyectables) necesarios para el entorno de prueba.
- El método configureTestingModule se suele invocar dentro de un método beforeEach de modo que TestBed pueda restablecer el estado base antes de cada ejecuciones de pruebas.

© JMA 2020. All rights reserved

Preparación de la prueba

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';

import { MyComponent } from './my.component';

describe('Prueba de MyComponent', () => {
  let fixture: ComponentFixture<MyComponent>;
  let comp:  MyComponent;
  let de:   DebugElement;
  let tag:  HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({ declarations: [ MyComponent ], });
    // ...
  })

  it('debe ...', () => {
    // Pruebas
  })
})
```

© JMA 2020. All rights reserved

Pruebas poco profundas

- En TestBed.configureTestingModule hay que declarar todas las dependencias del componente a probar: otros componentes, pipes y directivas propias, proveedores de los servicios utilizados, incluso los módulos importados.

```
TestBed.configureTestingModule({  
  imports: [MyCoreModule],  
  declarations: [MyComponent, OtherComponent, MyPipe, MyDirective, ...],  
})
```

- Agregando NO_ERRORS_SCHEMA (en @angular/core) a los metadatos del esquema del módulo de prueba se indica al compilador que ignore los elementos y atributos no reconocidos. Ya no es necesario declarar los elementos de plantilla irrelevantes.

```
TestBed.configureTestingModule({  
  declarations: [ MyComponent ],  
  schemas: [ NO_ERRORS_SCHEMA ]  
});
```

© JMA 2020. All rights reserved

ComponentFixture y DebugElement

- Un ComponentFixture es un contexto (fixture) que envuelve el componente creado en el entorno de prueba.
`fixture = TestBed.createComponent(MyComponent);`
- El fixture proporciona acceso a si mismo, a la instancia del componente y al envoltorio del elemento DOM del componente.
`comp = fixture.componentInstance; // instancia del componente`
- El DebugElement es un envoltorio del elemento DOM del componente o de los elementos localizados.
`de = fixture.debugElement;`
`tag = de.nativeElement;`

NOTA: Una vez ejecutado el método `createComponent` se cierra la configuración del `TestBed`, si se intenta cambiar la configuración dará un error.

© JMA 2020. All rights reserved

Consultar DebugElement

- La clase By es una utilidad Angular de pruebas para consultar el árbol del DOM. El método estático By.css utiliza un selector CSS estándar para generar un predicado (función que devuelve un valor booleano) que filtra de la misma manera que un selector de jQuery.
- Un predicado de consulta recibe un DebugElement y devuelve true si el elemento cumple con los criterios de selección.
- La clase DebugElement dispone de métodos para, utilizando una función de predicado, buscar en todo el árbol DOM del fixture:
 - El método query devuelve el primer elemento que satisface el predicado.
 - El método queryAll devuelve una matriz de todos los DebugElement que satisfacen el predicado.
- La propiedad nativeElement de DebugElement obtiene el elemento DOM.

```
let tag: HTMLElement = fixture.debugElement.query(By.css('#myId')).nativeElement;
```

© JMA 2020. All rights reserved

Consultas By

- Recuperar un componente con elemento HTML:

```
tag = fixture.debugElement.query(By.css('my-component'));
```
- Recuperar el valor de la primera etiqueta:

```
it('should render title in a h1 tag', waitForAsync(() => {
  fixture.detectChanges();
  const tag = fixture.debugElement.query(By.css('h1'));
  expect(tag.nativeElement.textContent).toContain('Welcome to app!!');
}););
```
- Recuperar los elementos de un listado:

```
it('renders the list on the screen', () => {
  fixture.detectChanges();
  const li = fixture.debugElement.queryAll(By.css('li'));
  expect(li.length).toBe(2);
});;
```

© JMA 2020. All rights reserved

triggerEventHandler

- El método DebugElement.triggerEventHandler permite simular que el elemento ha lanzado un determinado evento.
`fixture.debugElement.triggerEventHandler('click', eventData);`
- Según sea el evento, eventData representa el objeto event del DOM o el valor emitido. En algunos casos el eventData es obligatorio y con una determinada estructura.
- Dado que los eventos están vinculados a comandos, suele ser preferible invocar directamente a los métodos comando de la clase componente.
- Puede ser necesario para probar la interacción de determinadas directivas con el elemento y para probar las vinculaciones @HostListener.

© JMA 2020. All rights reserved

Detección de cambios

- La prueba puede decir a Angular cuándo realizar la detección de cambios, lo que provoca el enlace de datos y la propagación de las propiedades al elemento DOM.
`fixture.detectChanges()`
- Cuando se desea que los cambios se propaguen automáticamente sin necesidad de invocar `fixture.detectChanges()`:

```
import { ComponentFixtureAutoDetect } from '@angular/core/testing';
// ...
TestBed.configureTestingModule({
  declarations: [ MyComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```
- El servicio ComponentFixtureAutoDetect responde a las actividades asíncronas como la resolución de la promesa, temporizadores y eventos DOM. Pero una actualización directa, síncrona de una propiedad de componente es invisible. La prueba debe llamar `fixture.detectChanges()` manualmente para desencadenar otro ciclo de detección de cambios.

© JMA 2020. All rights reserved

Inyección de dependencias

```
TestBed.configureTestingModule({  
  declarations: [ MyComponent ],  
  providers: [ MyService ]  
});
```

- Un componente bajo prueba no tiene por qué ser inyectado con servicios reales, por lo general es mejor si son dobles de pruebas (stubs, fakes, spies o mocks), dado que el propósito de la especificación es probar el componente y no el servicio o servicios reales que pueden ser el origen del error.
 providers: [{provide: MyService, useValue: MyServiceFake },
 { provide: Router, useClass: RouterStub}]
- Si la prueba necesita tener acceso al servicio inyectado, la forma más segura es obtener el servicio desde el fixture.
 srv = fixture.debugElement.injector.get(MyService);
- También se puede obtener el servicio desde TestBed:
 srv = TestBed.inject(MyService);

© JMA 2020. All rights reserved

inject

- La función inject es una de las utilidades Angular de prueba.
 - Inyecta servicios en la función especificación donde se los puede alterar, espiar y manipular.
 - La función inject tiene dos parámetros:
 - Una matriz de tokens de inyección de dependencias Angular.
 - Una función de prueba cuyos parámetros corresponden exactamente a cada elemento de la matriz de tokens de inyección.
- ```
it('demo inject', inject([Router], (router: Router) => {
 // ...
}));
```
- La función inject utiliza el inyector del módulo TestBed actual y sólo puede devolver los servicios proporcionados a ese nivel. No devuelve los servicios de los proveedores de componentes.

**NOTA:** Una vez ejecutado el método `createComponent` se cierra la configuración del TestBed, si se intenta cambiar la configuración dará un error.

© JMA 2020. All rights reserved

## Sobre escritura de @Component

- En algunos casos, sobre todo en la inyección de dependencias a nivel de componentes es necesario sobreescibir la definición del componente.
- La estructura MetadataOverride establece las propiedades de @Component a añadir, modificar o borrar:  
`type MetadataOverride = { add?: T; remove?: T; set?: T; };`
- Donde T son las propiedades de la anotación:  
`selector?: string;  
template?: string; ó templateUrl?: string;  
providers?: any[];`  
...

© JMA 2020. All rights reserved

## Sobre escritura de @Component

- El método overrideComponent recibe el componente a modificar y los metadatos con las modificaciones:

```
TestBed.configureTestingModule({
 declarations: [MyComponent, MyChildComponent],
 providers: [{ provide: Router, useClass: RouterStub}]
})
.overrideComponent(MyChildComponent, {
 set: {
 providers: [
 { provide: MyService, useClass: MyServiceSpy }
]
 }
});
```

© JMA 2020. All rights reserved

## Creación asíncrona

- Si el componente tiene archivos externos de plantillas y CSS, que se especifican en las propiedades templateUrl y styleUrls, supone un problema para las pruebas.
- El método TestBed.createComponent es síncrono.
- Sin embargo, el compilador de plantillas Angular debe leer los archivos externos desde el sistema de archivos antes de que pueda crear una instancia de componente. Eso es una actividad asíncrona.
- El método compileComponents devuelve una promesa para que se puedan realizar tareas adicionales inmediatamente después de que termine.
- La función waitForAsync es una de las utilidades Angular de prueba que esconde la mecánica de ejecución asíncrona, envuelve una función de especificación en una zona de prueba asíncrona, la prueba se completará automáticamente cuando se finalicen todas las llamadas asíncronas dentro de esta zona.

© JMA 2020. All rights reserved

## Preparación de la prueba asíncrona

```
import { TestBed, waitForAsync } from '@angular/core/testing';
// ...
describe('AppComponent', () => {
 beforeEach(waitForAsync(() => {
 TestBed.configureTestingModule({
 declarations: [AppComponent],
 }).compileComponents();
 }));
 it('should create the app', waitForAsync(() => {
 const fixture = TestBed.createComponent(AppComponent);
 const app = fixture.debugElement.componentInstance;
 expect(app.toBeTruthy());
 }));
});
```

© JMA 2020. All rights reserved

## Inyección de servicios asíncronos

- Muchos servicios obtienen los valores de forma asíncrona. La mayoría de los servicios de datos hacen una petición HTTP a un servidor remoto y la respuesta es necesariamente asíncrona.
- Salvo cuando se estén probando los servicios asíncronos, las pruebas no deben hacer llamadas a servidores remotos. Las pruebas deberían emular este tipo de llamadas.
- Disponemos de las siguientes técnicas:
  - Sustituir el servicio asíncrono por un servicio síncrono.
  - Sustituir el método asíncrono por un espía.
  - Crear una zona de pruebas asíncrona.
  - Crear una falsa zona de pruebas asíncrona.

© JMA 2020. All rights reserved

## Espías

- Los espías de Jasmine permiten interceptar y sustituir métodos de los objetos.
- Mediante el espía se sustituye el método asíncrono de tal manera que cualquier llamada al mismo recibe una promesa resuelta de inmediato con un valor de prueba (stub).
- El espía no invoca el método real, por lo que no entra en contacto con el servidor.
- En lugar de crear un objeto de servicio sustituto, se inyecta el verdadero servicio y se sustituye el método crítico con un espía Jasmine.

© JMA 2020. All rights reserved

# Espías

```
beforeEach(() => {
 TestBed.configureTestingModule({
 declarations: [MyComponent],
 providers: [MyService],
 });
 fixture = TestBed.createComponent(MyComponent);
 comp = fixture.componentInstance;
 srv = fixture.debugElement.injector.get(MyService);
 spy = spyOn(srv, 'myAsyncMethod')
 .and.returnValue(Promise.resolve('result value'));
 // ...
});
it('Prueba asíncrona con espia', () => {
 // ...
 fixture.detectChanges();
 expect(...).matcher(...);
 expect(spy.calls.any()).toBe(true, 'myAsyncMethod called');
 expect(spy.calls.count()).toBe(1, 'stubbed method was called once');
 expect(myService.myAsyncMethod).toHaveBeenCalled();
});
```

© JMA 2020. All rights reserved

# whenStable

- En algunos casos, la prueba debe esperar a que la promesa se resuelva en el siguiente ciclo de ejecución del motor de JavaScript.
- En este escenario la prueba no se tiene acceso directo a la promesa devuelta por la llamada del método asíncrono dado que está encapsulada en el interior del componente, inaccesibles desde la superficie expuesta.
- Afortunadamente, la función `waitForAsync` genera una zona de prueba asíncrona, que intercepta todas las promesas emitidas dentro de la llamada al método asíncrono sin importar dónde se producen.
- El método `ComponentFixture.whenStable` devuelve su propia promesa que se resuelve cuando todas las actividades asíncronas pendientes dentro de la prueba se han completado (cuando sea estable).

```
it('Prueba asíncrona cuando sea estable', waitForAsync(() => {
 fixture.detectChanges();
 fixture.whenStable().then(() => {
 fixture.detectChanges();
 expect(...).matcher(...);
 });
}));
```

© JMA 2020. All rights reserved

## fakeAsync

- La función fakeAsync, otra de las utilidades Angular de prueba, es una alternativa a la función waitForAsync. La función fakeAsync permite un estilo de codificación secuencial mediante la ejecución del cuerpo de prueba en una zona especial de ensayo propia de fakeAsync, haciendo que la prueba aparezca como si fuera síncrona.
- Se apoya en la función tick (), que simula el paso del tiempo hasta que todas las actividades asíncronas pendientes concluyen, similar al wait en concurrencia. Sólo se puede invocar dentro del cuerpo de fakeAsync, no devuelve nada, no hay promesa que esperar.

```
it('Prueba asíncrona cuando con fakeAsync', fakeAsync(() => {
 fixture.detectChanges();
 tick();
 fixture.detectChanges();
 expect(...).matcher(...);
}));
```

© JMA 2020. All rights reserved

## Componentes contenidos

- Para probar los componentes simulando que están contenidos en una plantilla es necesario crear un componente Angular para la prueba (wrapper):

```
@Component({
 template: `<my-component [myInput]="MyInput"
 (myOutput)="onMyOutput($event)"></my-component>`
})
class TestHostComponent {
 @ViewChild(MyComponent) myComponent: MyComponent;
 MyInput: any = null;
 MyOutput: any;
 onMyOutput(rslt: any) { this.MyOutput = rslt; }
}
```

© JMA 2020. All rights reserved

# Componentes contenidos

- Para posteriormente instanciarlo:

```
beforeEach(waitForAsync(() => {
 TestBed.configureTestingModule({
 declarations: [MyComponent, TestHostComponent],
 }).compileComponents();
}));

beforeEach(() => {
 // create TestHostComponent instead of MyComponent
 fixture = TestBed.createComponent(TestHostComponent);
 testHost = fixture.componentInstance;
 tag = fixture.debugElement.query(By.css('my-component'));
 fixture.detectChanges(); // trigger initial data binding
});
```

© JMA 2020. All rights reserved

# Entradas y Salidas

- Entrada: Se modifican las entradas a través del componente de pruebas y se comprueba que las modificaciones se reflejan en el componente contenido.

```
it('input test', () => {
 testHost.MyInput = '666';
 fixture.detectChanges();
 expect(testHost.myComponent.getInit()).toBe('666');
});
```

- Salida: Se interactúa con el componente contenido para que se disparen los eventos de salida y se comprueba en el componente de pruebas que las modificaciones se han reflejado en el.

```
it('output test', () => {
 testHost.myComponent.exec();
 fixture.detectChanges();
 expect(testHost.MyOutput).toBe('666');
});
```

© JMA 2020. All rights reserved

# Pruebas de observables

- Para poder probar los observables es necesario:
  - Crear una zona asíncrona
  - Convertir el observable en una promesa
  - Definir las expectativas dentro del .then de la promesa.

```
import 'rxjs/add/operator/toPromise';

it('get http', waitForAsync(inject([HttpClient], (http: HttpClient) => {
 http.get(url)
 .toPromise().then(
 data => { expect(data).toBeTruthy(); },
 err => { fail(); }
);
})));
```

© JMA 2020. All rights reserved

# Pruebas de observables

- Para poder probar los observables mediante devolución de llamadas:

```
it('callback', (done: DoneFn) => inject([ValueService], (service) => {
 service.subscribe({
 next: data => { expect(data).toBe(0); done(); },
 error: () => fail()
 });
}));
```
- Se puede utilizar el marco de pruebas suministrado por RxJS.

```
import { TestScheduler } from 'rxjs/testing';
:
it('decrement contador', () => {
 let obs$ = service.getObservable();
 service.operate();
 testScheduler.run(helpers => {
 const { expectObservable } = helpers;
 expectObservable(obs$).toBe('a', {a:-1});
 });
});
```

© JMA 2020. All rights reserved

# Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Eso quiere decir que si una unidad tiene dependencias hay que reemplazarlas por mocks.
- Un ejemplo de mock de un servicio sería:

```
class MyServiceSpy {
 getData = jasmine.createSpy('getData').and.callFake(() => {
 return of([{ id: 0, name: 'Uno' }, { id: 1, name: 'Dos' }]);
 });
}
```
- Para crear las especificación:

```
it('fetches all data', () => {
 // ...
 expect(instance.names.length).toBe(2);
 expect(MyService.getData).toHaveBeenCalled();
});
```

© JMA 2020. All rights reserved

# Doble de prueba Observable

```
import { of, Observable } from 'rxjs';

export class DAOServiceMock {
 constructor(private listado: Array<any>) {}
 query(): Observable<any> { return of(this.listado); }
 get(id: number) { return of(this.listado[0]); }
 add(item: any) { return of(item); }
 change(id: number, item: any) { return of(item); }
 remove(id: number) { return of(id); }
}

{provide: MyDAOService, useValue: new DAOServiceMock([{ id: 0, name: 'Uno' }, { id: 1, name: 'Dos' }])}
```

© JMA 2020. All rights reserved

## Prueba de peticiones HTTP

- La biblioteca de pruebas HTTP de Angular está diseñada siguiendo un patrón de pruebas donde la especificación empieza haciendo las solicitudes.
- Después de eso, las pruebas esperan a que ciertas solicitudes hayan sido o no realizadas, se cumplan determinadas afirmaciones contra esas solicitudes y, finalmente, se proporcionan respuestas "descargando" cada solicitud esperada, lo que puede activar más solicitudes nuevas, etc.
- Al terminar, se pueden verificar que la aplicación no ha hecho peticiones inesperadas.
- Para disponer de la biblioteca de pruebas HTTP:  
imports: [ HttpClientTestingModule ],

© JMA 2020. All rights reserved

## Prueba de peticiones HTTP

- El módulo instala un mock que sustituye el acceso real al servidor.
- HttpTestingController es el controlador que se inyecta en las pruebas, permite la inspección y el volcado de las solicitudes.

```
it('query', inject([DAOService, HttpTestingController], (dao: DAOService, httpMock: HttpTestingController) => {
 dao.query().subscribe({
 next: data => { expect(data.length).toEqual(2); },
 error: data => { fail(); }
 });
 const req = httpMock.expectOne('http://localhost:4321/data');
 expect(req.request.method).toEqual('GET');
 req.flush([{ name: 'Data' }, { name: 'Data2' }]);
 httpMock.verify();
}));
```

© JMA 2020. All rights reserved

# Enrutado

- Para disponer del enrutador se utilizará el módulo:

```
imports: [// ...
 RouterTestingModule.withRoutes([
 {path: '', component: HomeCmp},
 {path: 'simple', component: SimpleCmp}
])]
```

- Para crear y registrar un sustituto del Router:

```
class RouterStub {
 navigateByUrl(url: string) { return url; }
 navigate(commands: Array<any>) { return url; }
}
```

- Para interceptar las llamadas al Router:

```
it('Demo ROUTER', inject([Router], (router: Router) => { // ...
 const spy = spyOn(router, 'navigateByUrl');
 // ...
 const navArgs = spy.calls.first().args[0];
 expect(navArgs).toBe(...);
}));
```

© JMA 2020. All rights reserved

<http://www.protractortest.org/>

**TEST E2E**

© JMA 2020. All rights reserved

# Introducción

- A medida que las aplicaciones crecen en tamaño y complejidad, se vuelve imposible depender de pruebas manuales para verificar la corrección de las nuevas características, errores de captura y avisos de regresión.
- Las pruebas unitarias son la primera línea de defensa para la captura de errores, pero a veces las circunstancias requieren la integración entre componentes que no se pueden capturar en una prueba unitaria.
- Las pruebas de extremo a extremo (E2E: end to end) se hacen para encontrar estos problemas.
- El equipo de Angular ha desarrollado Protractor que simula las interacciones del usuario con el interfaz (navegador) y que ayudará a verificar el estado de una aplicación Angular.
- Protractor es una aplicación Node.js para ejecutar pruebas de extremo a extremo que también están escritas en JavaScript y se ejecutan con el propio Node.
- Protractor utiliza WebDriver para controlar los navegadores y simular las acciones del usuario.

© JMA 2020. All rights reserved

# Introducción

- Protractor utiliza Jasmine para su sintaxis prueba.
- Al igual que en las pruebas unitarias, el archivo de pruebas se compone de uno o más bloques describe de it que describen los requisitos de su aplicación.
- Las bloques it están hechos de comandos y expectativas.
- Los comandos indican a Protractor que haga algo con la aplicación, como navegar a una página o hacer clic en un botón.
- Las expectativas indican a Protractor afirmaciones sobre algo acerca del estado de la aplicación, tales como el valor de un campo o la URL actual.
- Si alguna expectativa dentro de un bloque it falla, el ejecutor marca en it como "fallido" y continúa con el siguiente bloque.
- Los archivos de prueba también pueden tener bloques beforeEach y afterEach, que se ejecutarán antes o después de cada bloque it, independientemente de si el bloque pasa o falla.

© JMA 2020. All rights reserved

# Instalación

- Se utiliza npm para instalar globalmente Protractor:
  - npm install -g protractor
- Esto instalará dos herramientas de línea de comandos, protractor y WebDriver-manager, para asegurarse de que está funcionando.
  - protractor –versión
- El WebDriver-Manager es una herramienta de ayuda para obtener fácilmente una instancia de un servidor en ejecución Selenium. Para descargar los binarios necesarios:
  - webdriver-manager update
- Para poner en marcha el servidor:
  - webdriver-manager start
- Las pruebas Protractor enviarán solicitudes a este servidor para controlar un navegador local, el servidor debe estar en funcionamiento durante todo el proceso de pruebas.
- Se puede ver información sobre el estado del servidor en:
  - <http://localhost:4444/wd/hub>

© JMA 2020. All rights reserved

# Configuración y Ejecución

- Se configura un fichero con las pruebas a realizar:

```
// Fichero: e2e.conf.js
exports.config = {
 framework: 'jasmine',
 seleniumAddress: 'http://localhost:4444/wd/hub',
 specs: ['test/*.e2e.js'],
 multiCapabilities: [
 //{{ browserName: 'firefox' },
 { browserName: 'chrome' }
]
};
```
- Se lanzan las pruebas (con Selenium Server en ejecución):
  - protractor e2e.conf.js

© JMA 2020. All rights reserved

# Elementos Globales

- browser: Envoltura alrededor de una instancia de WebDriver, utilizado para la navegación y la información de toda la página.
  - El método browser.get carga una página.
  - Protractor espera que Angular esté presente la página, por lo que generará un error si la página que está intentando cargar no contiene la biblioteca Angular.
- element: Función de ayuda para encontrar e interactuar con los elementos DOM de la página que se está probando.
  - La función element busca un elemento en la página.
  - Se requiere un parámetro: una estrategia de localización del elemento dentro de la página.
- by: Colección de estrategias elemento localizador.
  - Por ejemplo, los elementos pueden ser encontrados por el selector CSS, por el ID, por el atributo ng-model, ...
- protractor: Espacio de nombres de Angular que envuelve el espacio de nombres WebDriver.
  - Contiene variables y clases estáticas, tales como protractor.Key que se enumera los códigos de teclas especiales del teclado.

© JMA 2020. All rights reserved

# Visión de conjunto

- Protractor exporta la función global element, que con un localizador devolverá un ElementFinder.
- Esta función recupera un solo elemento, si se necesita recuperar varios elementos, la función element.all obtiene la colección de elementos localizados.
- El ElementFinder tiene un conjunto de métodos de acción, tales como click(), getText(), y sendKeys().
- Esta es la forma principal para interactuar con un elemento (etiqueta) de la página y obtener información de respuesta de él.
- Cuando se buscan elementos en Protractor todas las acciones son asíncronas:
  - Por debajo, todas las acciones se envían al navegador mediante el protocolo SON Webdriver Wire Protocol.
  - El navegador realiza la acción tal y como un usuario lo haría de forma nativa o manual.

© JMA 2020. All rights reserved

# Localizadores

- Un localizador de Protractor dice cómo encontrar un cierto elemento DOM.
- Los localizadores más comunes son:
  - by.css('.myclass')
  - by.id('myid')
  - by.model('name')
  - by.binding('bindingname')
- Los localizadores se pasan a la función element:
  - var tag = element(by.css('some-css'));
  - var arr = element.all(by.css('some-css'));
- Aunque existe una notación abreviada para CSS similar a jQuery:
  - var tag = \$('some-css');
- Para encontrar subelementos o listas de subelementos:
  - var uno = element(by.css('some-css')).element(by.tagName('tag-within-css'));
  - var varios = element(by.css('some-css')).all(by.tagName('tag-within-css'));

© JMA 2020. All rights reserved

# ElementFinder

- La función element() devuelve un objeto ElementFinder.
- El ElementFinder sabe cómo localizar el elemento DOM utilizando el localizador que se pasa como un parámetro, pero en realidad no lo ha hecho todavía.
- No va a ponerse en contacto con el navegador hasta que un método de acción sea llamado.
- ElementFinder permite invocar acciones como si se produjesen directamente en el navegador.
- Dado que todas las acciones son asíncronas, todos los métodos de acción devuelven una promesa.
- Las acciones sucesivas se encolan y se mandan la navegador ordenadamente.
- Para acciones que deban esperar se usan las promesas:

```
element(by.model('nombre')).getText().then(function(text) {
 expect(text).toBe("MUNDO");
});
```

© JMA 2020. All rights reserved

## La prueba

```
describe('Primera prueba con Protractor', function() {
 it('introducir nombre y saludar', function() {
 browser.get('http://localhost:4200/');
 var txt = element(by.model('vm.nombre'));
 txt.clear();
 txt.sendKeys('Mundo');
 browser.sleep(5000);
 element(by.id('btnSaluda')).click();
 expect(element(by.binding('vm.msg')).getText()).toEqual('Hola Mundo');
 browser.sleep(5000);
 });
});
```

© JMA 2020. All rights reserved

## Selenium

- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web.
- Como principales herramientas Selenium cuenta con:
  - Selenium IDE: una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
  - Selenium Core: API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, Php y Python.
  - WebDriver: interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
  - Selenium Grid: Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias maquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

© JMA 2020. All rights reserved

## Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox y Chrome.
- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese.
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
  - **Acciones:** son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait" que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
  - **Descriptores de acceso:** examinan el estado de la página y almacenan los resultados en variables.
  - **Aserciones:** son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

© JMA 2020. All rights reserved

## Selenium IDE

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- ~~Permite exportar los casos de prueba a Java, C# y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.~~
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

© JMA 2020. All rights reserved

# WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
 System.setProperty("webdriver.chrome.driver", "C:/Archivos/.../chromedriver.exe");
}

@Before
public void setUp() throws Exception {
 driver = new ChromeDriver();
 baseUrl = "http://localhost/";
 driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}

@Test
public void testLoginOK() throws Exception {
 driver.get(baseUrl + "/login.php");
 driver.findElement(By.id("login")).sendKeys("admin");
 driver.findElement(By.id("password")).sendKeys("admin");
 driver.findElement(By.cssSelector("input[type='submit']")).click();
 try {
 assertEquals("", driver.findElement(By.cssSelector("img[title='Main Menu']")).getText());
 } catch (Error e) {
 verificationErrors.append(e.toString());
 }
}
```

## Maven

```
<dependency>
<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId>
<version>3.13.0</version>
</dependency>
```

© JMA 2020. All rights reserved

## Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (<https://nodejs.org>)
- Instalar CLI
  - npm install -g selenium-side-runner
- Instalar los WebDriver:
  - Crear una carpeta y referenciarla en el PATH del sistema.
  - Descargar los drivers (<https://www.seleniumhq.org/download/>)
  - Copiarlos a la carpeta creada.
- Para ejecutar las suites de pruebas:
  - selenium-side-runner project.side project2.side \*.side
- Para ejecutar en diferentes navegadores:
  - selenium-side-runner \*.side -c "browserName=Chrome"
  - selenium-side-runner \*.side -c "browserName=firefox"

© JMA 2020. All rights reserved

# Proceso de Prueba

- Descomponer la aplicación web existente para identificar qué probar
- Identificar con qué navegadores probar
- Elige el mejor lenguaje para ti y tu equipo.
- Configura Selenium para que funcione con cada navegador que te interese.
- Escriba pruebas de Selenium mantenibles y reutilizables que serán compatibles y ejecutables en todos los navegadores.
- Cree un circuito de retroalimentación integrado para automatizar las ejecuciones de prueba y encontrar problemas rápidamente.
- Configura tu propia infraestructura o conéctate a un proveedor en la nube.
- Mejora drásticamente los tiempos de prueba con la paralelización
- Mantente actualizado en el mundo Selenium.

© JMA 2020. All rights reserved

## Selenium

- <http://www.seleniumhq.org/>
- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web.
- Como principales herramientas Selenium cuenta con:
  - Selenium IDE: una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
  - Selenium Core: API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, Php y Python.
  - WebDriver: interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
  - Selenium Grid: Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias máquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

© JMA 2020. All rights reserved

## Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox.
- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese.
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
  - **Acciones:** son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait" que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
  - **Descriptores de acceso:** examinan el estado de la página y almacenan los resultados en variables.
  - **Aserciones:** son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

© JMA 2020. All rights reserved

## Selenium IDE

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- Permite exportar los casos de prueba a Java, C# y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

© JMA 2020. All rights reserved

# WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
 System.setProperty("webdriver.chrome.driver", "C:/Archivos/.../chromedriver.exe");
}

@Before
public void setUp() throws Exception {
 driver = new ChromeDriver();
 baseUrl = "http://localhost/";
 driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}

@Test
public void testLoginOK() throws Exception {
 driver.get(baseUrl + "/login.php");
 driver.findElement(By.id("login")).sendKeys("admin");
 driver.findElement(By.id("password")).sendKeys("admin");
 driver.findElement(By.cssSelector("input[type='submit']")).click();
 try {
 assertEquals("", driver.findElement(By.cssSelector("img[title='Main Menu']")).getText());
 } catch (Error e) {
 verificationErrors.append(e.toString());
 }
}
```

© JMA 2020. All rights reserved