Angular

Es una plataforma y un framework para crear aplicaciones cliente en HTML y TypeScript. Angular está escrito en TypeScript. Implementa funciones básicas y opcionales como un conjunto de bibliotecas TypeScript que se importan en las aplicaciones. Angular aprovecha la tecnología de los componentes web o *'web components'* y el *'Shadow DOM'* para apoyar el desarrollo impulsado por componentes.

Los elementos más básicos de construcción de una aplicación Angular son los *NgModules*, que proporcionan un contexto de compilación para los componentes. Los *NgModules* agrupan el código relacionado en conjuntos funcionales; una aplicación Angular se define por un conjunto de *NgModules*. Una aplicación siempre tiene al menos un módulo raíz que permite el bootstrapping, y normalmente tiene muchos más módulos de características.

Los componentes definen **vistas**, que son conjuntos de elementos de pantalla entre los que Angular puede elegir y modificar según la lógica y los datos de su programa.

Los componentes utilizan **servicios** que proporcionan funciones específicas que no están directamente relacionadas con las vistas. Los proveedores de servicios pueden ser inyectados en los componentes como dependencias, haciendo que su código sea modular, reutilizable y eficiente.

Tanto los componentes como los servicios son simplemente clases, con decoradores que marcan su tipo y proporcionan metadatos que indican a Angular cómo utilizarlos.

Los metadatos para una clase de componente la asocian con una plantilla o *'template'* que define una vista. Una plantilla combina HTML ordinario con directivas de Angular y *'binding markup'* que permiten a Angular modificar el HTML antes de renderizarlo para mostrarlo.

Los metadatos para una clase de servicio proporcionan la información que Angular necesita para ponerla a disposición de los componentes a través de la inyección de dependencia (DI).

Los componentes de una aplicación suelen definir muchas vistas, ordenadas jerárquicamente. Angular proporciona el servicio de enrutado para ayudarle a definir rutas de navegación entre las vistas. El enrutador proporciona sofisticadas capacidades de navegación dentro del navegador.

Primeros pasos

Prerequisitos

Angular requiere Node.js versión 10.0.9 o superior. Podemos instalar Node.js descargándolo de su página oficial.

Cuando tengamos instalado *node.js* o si ya lo teníamos instalado podemos comprobar si está correctamente configurado escribiendo en el terminal `node -v`. Si está todo correcto nos mostrará la versión de *node.js*.

Cuando se instala *node.js* también se instala *NPM*. Para comprobar si está correctamente instalado escribimos en el terminal `npm -v` lo que nos mostrará la versión instalada.

Para desarrollar en Angular podemos usar simplemente Javascript, pero se recomienda usar TypeScript, dado que Angular está desarrollado con TypeScript y todos los ejemplos y código que se encuentra en la web estará escrito en TypeScript. Para instalar Typescript se utiliza también la herramienta *npm*. Abrimos un terminal/consola y ejecutamos `npm install -g typescript`.

```
//Summary
node -v
npm -v
```

```
npm install -g typescript
tsc --version
```

Instalación de *'Angular CLI'*

'Angular CLI' o simplemente *CLI* es la herramienta de línea de comandos estándar para crear, depurar y publicar aplicaciones Angular.

Para instalar 'Angular CLI' abrimos un terminal/consola y ejecutamos el comando `npm install -g @angular/cli` lo que instala 'Angular CLI' de forma global lo que nos permitirá usar la herramienta desde cualquier directorio.

La sintaxis para instalar Angular (@angular/paquete) es una nueva característica de *npm* llamada **'scoped packages'**. Permite agrupar los paquetes en una misma carpeta.

Comprobamos la versión instalada ejecutando en un terminal/consola `ng version`. La ayuda está disponible tanto de modo general (`ng help`) como para cada comando de la herramiento (`ng new --help` o por ejemplo `ng generate --help`).

```
npm install -g @angular/cli   // Instalar Angular CLI
ng version
ng help                       // Ver ayuda general
ng generate --help            // Ayuda de un comando en concreto
```

Crear un *'workspace'* y una aplicación inicial

Las aplicaciones Angular se desarrollan en el contexto de una espacio de trabajo o 'workspace'. Un espacio de trabajo puede contener múltiples aplicaciones y bibliotecas. La aplicación inicial creada con el comando `ng new` está en el nivel top del espacio de trabajo. Cuando generamos aplicaciones o bibliotecas adicionales, éstas se crean dentro de la subcarpeta `projects/` .

Para crear un nuevo espacio de trabajo y una aplicación inicial dentro de este nuevo espacio de trabajo, abrimos un terminal y escribimos:

```
ng new my-app  // Cambiando 'my-app' por el nombre de nuestra nueva aplicación
```

El comando `ng new` nos solicitará información para configurar la aplicación inicial. Lo más habitual es usar la configuración que viene por defecto. Para más información podemos consultar la documentación del comando `ng new` .

'Angular CLI' instalará todas las bibliotecas y dependencias necesarias, lo que se podrá demorar varios minutos.

Finalmente tendremos una aplicación inicial completamente funcional y las configuraciones necesarias para su depuración, pruebas y ejecución.

Por defecto la aplicación se crea con el prefijo `app` que se usará en todos los componentes. Puede personalizarse usando el modificador `--prefix` , como por ejmplo `ng new --prefix sigma` .

Ejecutar la aplicación

'Angular CLI' incluye un servidor, para que pueda crear y servir su aplicación fácilmente a nivel local.

Entramos en el directorio de la apliación y lanzamos el servidor con `ng serve --open` . Con el parámetro `--open` le indicamos que abra el navegador y cargue la página http://localhost:4200:

```
cd my-app
ng serve --open // Abre el navegador directamente
```

El comando `ng serve` inicia el servidor, observa el código fuente, reconstruye automáticamente la aplicación cuando detecta algún cambio en el código y recarga la página en el navegador.

Estructura de una aplicación Angular

https://angular.io/guide/file-structure

Cuando ejecutamos el comando `ng new` se instalan las bibliotecas y dependencias necesarias en el nuevo *workspace* además del esqueleto funcional de una aplicación dentro de la carpeta `src/` . Esta aplicación se considera la aplicación principal o aplicación raíz. El directorio raíz del espacio de trabajo contiene todos los ficheros de configuración, etc.. necesarios para construir y servir la aplicación Angular.

La aplicación inicial creada es la apliación por defecto para todos los comandos lanzados a través de la CLI.

Para un espacio de trabajo que contendrá una sóla aplicación, la subcarpeta `src/` del espacio de trabajo contendrá los ficheros de código (lógica de la aplicación, datos y assets) de la aplicación raíz. Para espacios de trabajo de tipo *'multi-project'* cada proyecto estará en su propia carpeta dentro de la carpeta `projects/` .

Ficheros de configuración

https://angular.io/guide/file-structure#workspace-configuration-files

Todos los proyectos dentro del mismo espacio de trabajo o *'workspace'* comparten los ficheros de configuración que están en la raíz del espacio de trabajo. Estos ficheros de configuración tienen ámbito del espacio de trabajo y por tanto su configuración afecta a todos los proyectos.

Los archivos de configuración que están en la raíz del espacio de trabajo son los ficheros de configuración de la aplicación raíz. Para un espacio de trabajo de múltiples proyectos, los archivos de configuración específicos de cada proyecto estarán en la carpeta raíz de cada proyecto, dentro de `projects/project-name` .

El fichero `'package.json'` es el fichero estándar de *npm* donde se almacenan las dependencias de terceros que se utilizará para todos los proyectos del espacio de trabajo. Contiene las bibliotecas que necesita la aplicación para ejecutarse tanto en desarrollo como producción.

```
{
  "dependencies": {
    "@angular/core": "^7.2.0"
  },
  "devDependencies": {
    "@angular/cli": "7.2.0"
  }
}
```

Además de las dependencias el fichero `'package.json'` sirve para indicar información sobre el proyecto como el nombre, versión, nombre del autor, url del repositorio, etc.. y también como contenedor de scripts para automatizar tareas de operaciones rutinarias:

```
{
  "name": "my-app",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
```

```
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "devDependencies": {
    // ...
  }
}
```

El fichero `'tsconfig.json'` contiene los parámetros de configuración por defecto de TypeScript.

El fichero `angular.json` contiene los valores predeterminados de configuración de la CLI para todos los proyectos en el área de trabajo, incluidas las opciones de configuración para compilar, servir y probar las herramientas que utiliza la CLI

Los cambios en los ficheros de configuración no se recargan automáticamente. Hay que parar la servidor y volver a lanzarlo para que se carguen.

Carpetas y ficheros de la aplicación

https://angular.io/guide/file-structure#application-project-files

Dentro de la carpeta `/src` tenemos:

- La carpeta `app/` contiene los componentes Angular que componen la aplicación.
- La carpeta `environments/` contiene dos ficheros, y puede contener más, para cada entorno de distribución necesario. En código siempre importaremos el fichero base, pero durante la compilación el CLI lo sustituirá por el adecuado como por ejemplo `title = environment.appName + 'hello world ;-)';`
- La carpeta `assets/` contiene imágenes y otros archivos de activos que se copiarán tal cual cuando cree su aplicación.
- `index.html`: La página HTML principal que se sirve cuando alguien visita el sitio. La CLI agrega automáticamente todos los archivos JavaScript y CSS al crear su aplicación, por lo que normalmente no necesita agregar ninguna etiqueta `<script>` o `<link>` manualmente.
- `main.ts`: El principal punto de entrada para su aplicación. Compila la aplicación con el compilador JIT y arranca el módulo raíz de la aplicación (`AppModule`) para ejecutarse en el navegador. También puede usar el compilador AOT sin cambiar ningún código agregando el indicador --aot a los comandos de compilación y servicio de la CLI.

Dentro tenemos la carpeta `/src/app/` :

- `app.module.ts`: Define el módulo raíz, llamado `AppModule` , que le dice a Angular cómo ensamblar la aplicación. Inicialmente declara solo el `AppComponent` . A medida que agregue más componentes a la aplicación, deberá declararlos aquí.

Una aplicación Angular es un árbol de componentes cuyo componente raíz es llamado `AppComponent` :

- `app.component.ts`: Define la lógica para el componente raíz de la aplicación. La vista asociada con este componente raíz se convierte en la raíz de la jerarquía de vistas a medida que agrega componentes y servicios a su aplicación.
- `app.component.html`: Define la plantilla HTML asociada con el componente raíz `AppComponent`
- `app.component.css`: Define la hoja de estilo CSS básica para el componente raíz `AppComponent`

Arquitectura de una aplicación Angular

https://angular.io/guide/architecture
https://academia-binaria.com/base-aplicacion-angular/

Introducción a los Módulos

https://angular.io/guide/architecture-modules#introduction-to-modules
https://angular.io/guide/ngmodules

Las aplicaciones en Angular son modulares y Angular tiene su propio sistema de módulos llamado *'NgModules'*.

Los módulos son **contenedores para almacenar los componentes y servicios** de una aplicación. En Angular toda aplicación se puede ver como un árbol de módulos jerárquico y por tanto como mínimo tendrá un módulo raíz, que por convención se llama **'AppModule'** y reside en el fichero `app.module.ts`. A partir de ese módulo raíz se enlazan el resto de módulos, si los hubiera.

Los módulos pueden exportar funcionalidad que será importada y utilizada por otros módulos.

Un *'NgModule'* se define como una clase TypeScript decorada con la función decoradora `@NgModule()` que recibe un objeto con metadatos como único argumento. En las propiedades de ese objeto es donde se configura el módulo.

Este es un ejemplo de un módulo raíz `AppModule` que reside en el fichero `'app.module.ts'`:

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

@NgModule({
  declarations: [ AppComponent ], // Componentes, directivos y/o pipes que pertenecen al módulo
  imports: [ BrowserModule ],     // Módulos que se importan para poder utilizar su funcionalidad
  providers: [],                  // Los proveedores son creadores de servicios
  bootstrap: [AppComponent]       // El componente raíz principal que inicia la aplicación
})
export class AppModule { }
```

Para crear un módulo abrimos un terminal/consola y ejecutamos el comando:

```
ng generate module login // Substituir 'login' por el nombre del módulo
```

Esto creará el fichero `'login/login.module.ts'`. Para utilizar el nuevo módulo hay que añadirlo en la propiedad `imports:[]`, que es un array de punteros a otros módulos que está en el fichero `'app.module.ts'` que es el fichero del módulo raíz.

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

import { AppComponent } from './app.component';
import { LoginModule } from './login/login.module';

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

El módulo raíz siempre tiene un **componente raíz** que se crea cuando se crea el módulo. Los módulos pueden tener más de un componente aunque sólo uno será el componente raíz. El componente raíz se indica en `bootstrap: [AppComponent]` de forma

que cualquier componente con capacidad para actuar como punto de entrada debe indicarse en esta propiedad.

Introducción a los Componentes

https://angular.io/guide/architecture-components

Los componentes son los bloques básicos de construcción de las páginas web en Angular. Contienen una parte visual en HTML, que es una plantilla o *'template'* (la parte visual) y una funcional en una clase en Typescript (el controlador). La clase interactúa con la plantilla a través de una API de propiedades y métodos.

Ambas partes, la plantilla y el controlador definen lo que es una **vista**. Un componente puede contener una jerarquía de vistas, lo que permite construir interfaces visuales complejas y modificar, crear, añadir o destruir determinadas partes según convenga. Además, no estamos limitados a un sólo módulo, de tal forma que una jerarquía de vistas puede mezclar vistas definidas en componentes que están en diferentes módulos.

La sintaxis para crear nuevos componentes es:

```
ng generate component <name>
```

El componente raíz que se genera en el módulo raíz es el `AppComponent`. Según la configuración del CLI este componente puede haber sido creado en un sólo fichero o en hasta cuatro:

- el controlador en `app.component.ts`
- la vista en `app.component.html`
- el estilo en `app.component.css`
- las pruebas unitarias en `app.component.spec.ts`

Angular crea, actualiza y destruye componentes a medida que el usuario se mueve a través de la aplicación. La aplicación puede realizar acciones en cada momento de este ciclo de vida a través de los 'lifecycle hooks' como `ngOnInit()` al estilo de Android y los métodos `onCreate()`, `onStart()`, etc…

Los componentes, como el resto de artefactos en Angular, definen su lógica en **clases TypeScript decoradas** con una función específica. En este caso para que Angular sepa que es un componente la clase estará *'decorada'* con la función `@Component()` que recibe un objeto de metadatos que define el componente. Al igual que en los módulos, este objeto contiene las propiedades con las que se configura el componente.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: []
})
export class AppComponent {}
```

Los metadatos de un componente le indican a Angular dónde obtener los bloques de construcción principales que necesita para crear y presentar el componente y su vista. En particular, asocia una plantilla con el componente, ya sea directamente con código en línea o por referencia. Juntos, el componente y su plantilla describen una vista.

Los componentes definen nuevas etiquetas HTML para ser usados dentro de otros componentes. Excepcionalmente, el componente raíz se utiliza en la página `'index.html'`. El nombre de la nueva etiqueta se conoce como *selector*. En el caso del componente raíz la propiedad `selector: "app-root"` permite el uso de este componente con esta invocación `<app-root></app-root>`. Otros componentes definirán sus etiquetas HTML dentro de la propiedad `selector` y Angular creará e insertará una instancia del componente donde encuentre esa etiqueta incluido dentro de otros componentes creando una jerarquía.

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

La plantilla representa la parte visual del componente y le dice a Angular como renderizar el componente. De forma simplificada, o cuando tiene poco contenido, puede escribirse directamente en la propiedad `template` del objeto decorador. Pero es más frecuente encontrar la plantilla en su propio fichero html y referenciarlo como una ruta relativa en la propiedad `templateUrl`.

La propiedad `styles` y su gemela `stylesUrl` permiten asignar estilos CSS, SASS o LESS al componente. Estos estilos se incrustan durante la compilación en los nodos del DOM generado. Son exclusivos del componente y facilitan el diseño y maquetación granular de las aplicaciones.

En la clase del componente, que actua como controlador, nos encontraremos **la implementación de su funcionalidad**. Normalmente expondrá propiedades y métodos para ser consumidos e invocados de forma declarativa desde la vista.

Los componentes no deciden por sí mismos su visibilidad. Cuando un componente es generado se declara en un módulo contenedor en su propiedad `declares:[]`. Eso lo hace visible y utilizable por cualquier otro componente del mismo módulo. Para poder usarlo en otros módulos hay que exportarlo de forma explícita. Eso se hace en la propiedad `exports:[]` del módulo en el que se crea o indicarse con el flag `--export`.

Que un componente sea público es la primera condición para que se consuma fuera de su módulo. El siguiente paso es importarlo en la propiedad `imports:[]` del módulo. Como regla general, cuando en una plantilla se incruste otro componente, Angular lo buscará **dentro del propio módulo** en el que pretende usarse. Si no lo encuentra entonces lo buscará entre los componentes exportados por los módulos que hayan sido importados por el actual contenedor.

Los componentes privados suelen ser sencillos. A veces son creados para ser específicamente consumidos dentro de otros componentes. En esas situaciones interesa que sean privados y que generen poco ruido. Incluso, en casos extremadamente simples, si usamos el modificador `--flat` ni siquiera generan carpeta propia.

Cada módulo puede exportar sus propios componentes o los componentes de terceros, incluso un módulo completo, haciendo uso de la interesante **propiedad transitiva de los módulos**.

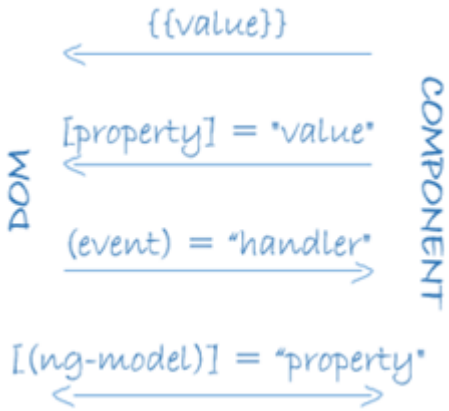
Sintaxis de las plantillas

Una plantilla contiene etiquetas HTML normales pero además contiene una sintaxis de plantillas propia qyue altera el HTML basado en la lógica de la aplicación, su estado y los datos del DOM.

La plantilla usan el *'data binding'* para coordinar los datos de la aplicación con los datos del DOM, *'pipes'* para transformar los datos antes de ser mostrados y *'directives'* para aplicar la lógica de la aplicación a lo que se muestra.

Data binding

Angular admite el enlace de datos bidireccional o *'two-way data binding'*, un mecanismo para coordinar las partes de una plantilla con las partes de un componente. Para indicar a Angular como tiene que conectar ambas partes y la dirección de la conexión, se utiliza un marcado especial que tiene hasta 4 formas:



	Markup	Nombre
DOM <------- Component	{{value}}	*'interpolation'*
DOM <------- Component	[property] = "value"	*'property binding'*
DOM -------> Component	(event) = "handler"	*'event binding'*
DOM <------> Component	[(ng-model)] = "property"	*'two-way binding'*

```
<li>{{hero.name}}</li>  <!-- Interpolation permite visualizar la propiedad 'hero.name' del componente en la etiqueta --

<app-hero-detail [hero]="selectedHero"</app-hero-detail> <!-- Pasa el valor del 'selectedHero' del padre al componente

<li (click)="selectedHero(hero)"></li> <!-- Llama al método 'selectedHero(hero)' cuando el usuario hace click sobre la

<input [(ngModel)]="hero.name"> <!-- two-way binding -->
```

En *'two-way binding'* un valor de propiedad de datos fluye al input desde el componente como con el enlace de propiedades o *'property binding'*. Los cambios del usuario también vuelven al componente, restableciendo la propiedad al último valor, como con el enlace de eventos o *'event binding'*. Por tanto es un enlace bidireccional que es la suma de ambos enlaces.

Pipes

Las *'pipes'* le permiten declarar transformaciones de valor de visualización en su plantilla HTML. Una clase con el decorador `@Pipe` define una función que transforma los valores de entrada en valores de salida para mostrar en una vista.

Angular tiene definidas por defecto varias *'pipes'* para trabajar con fechas o monedas. La lista completa se encuentra aquí.

Para especificar una transformación en HTML con las *'pipes'* se utiliza el *'pipe operator (|)'*:

```
<li>{{ interpolated value | pipe_name}}</li>
```

Las *'pipes'* se pueden encadenar, enviando la salida de una tubería para que sea transformada por otra tubería. Una *'pipe'* también puede tomar argumentos que controlan cómo realiza su transformación. Por ejemplo, puede pasar el formato deseado a la tubería de fecha:

```
<!-- Default format: output 'Jun 15, 2015'-->
 <p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
```

```
<p>The date is {{today | date:'fullDate'}}</p>

 <!-- shortTime format: output '9:43 AM'-->
 <p>The time is {{today | date:'shortTime'}}</p>
```

Directivas

Las plantillas en Angular son dinámicas. Cuando Angular las procesa, transforma el DOM de acuerdo con las instrucciones dadas por las directivas. Una directiva es una clase con un decorador `@Directive()`.

Un componente es técnicamente una directiva. Sin embargo, los componentes son tan distintivos y centrales para las aplicaciones de Angular que Angular define el decorador `@Component()`, que extiende el decorador `@Directive()` con características orientadas a plantillas.

Además de los componentes, hay otros dos tipos de directivas: **estructurales y de atributo**. Angular define una serie de directivas de ambos tipos, pero se pueden definir directivas propias utilizando el decorador `@Directive()`.

Las directivas estructurales alteran el diseño agregando, eliminando y reemplazando elementos en el DOM:

```
<li *ngFor="let hero of heroes"></li> <!-- *ngFor is an iterative -->

<app-hero-detail *ngIf="selectedHero"></app-hero-detail> <!-- *ngIf is a conditional -->
```

Las directivas de atributos alteran la apariencia o el comportamiento de un elemento existente. En las plantillas se ven como atributos HTML normales, de ahí el nombre.

La directiva `ngModel`, que implementa el enlace de datos bidireccional, es un ejemplo de una directiva de atributo. `ngModel` modifica el comportamiento de un elemento existente (generalmente `<input>`) estableciendo su propiedad de valor de visualización y respondiendo a eventos de cambio.

```
<input [(ngModel)]="hero.name">
```

Introducción a los Servicios y DI

https://angular.io/guide/architecture-services
https://angular.io/guide/dependency-injection
https://academia-binaria.com/servicios-inyectables-en-Angular

Un servicio es típicamente una clase con un propósito estrecho y bien definido. Debe hacer algo específico y hacerlo bien. Su propósito puede ser contener lógica de negocio, obtener datos de servidores externos, etc…

Angular distingue los componentes de los servicios para aumentar la modularidad y la reutilización. Al separar la funcionalidad relacionada con la vista de un componente de otros tipos de procesamiento, puede hacer que sus clases de componentes sean livianas y eficientes.

Idealmente, el trabajo de un componente es permitir la experiencia del usuario y nada más. Un componente debe presentar propiedades y métodos para el *'data binding'*, para mediar entre la vista (representada por la plantilla) y la lógica de la aplicación (que a menudo incluye alguna noción de modelo).

Un componente puede delegar ciertas tareas a los servicios, como obtener datos del servidor, validar la entrada del usuario o iniciar sesión directamente en la consola. Al definir tales tareas de procesamiento en una clase de servicio inyectable, pasan a estar a disposición de cualquier componente. Esto también permite más flexibilidad a las aplicaciones ya que podemos inyectar diferentes proveedores del mismo tipo de servicio, según corresponda en diferentes circunstancias.

Los servicios a su vez pueden depender de otros servicios.

```
// Ejemplo de servicio en Angular
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

La DI o **'Inyección de Dependencias'** está íntimamente conectado en el framework de Angular. Angular crea un *'inyector'* cuando se crea la aplicación. Este inyector se encarga de crear las dependencias y mantener un contenedor de dependencias que se reutilizarán cuando sea posible. También existe la figura del *'proveedor'* que es un objeto que le dice al inyector como obtener o crear la dependencia.

Para cualquier dependencia que necesite en su aplicación, debe registrar un proveedor con el inyector de la aplicación, para que el inyector pueda usar el proveedor para crear nuevas instancias. Para un servicio, el **proveedor suele ser la clase de servicio en sí misma**.

Los componentes consumen los servicios, lo que significa que se *inyectan* dinámicamente en el componente para que pueda ser utilizado. Una dependencia no tiene por qué ser únicamente un servicio. Puede ser una función, un valor, etc…

Para definir una clase como servicio se decora con `@Injectable()` y se provee de los metadatos necesarios para que Angular pueda inyectar el servicio en el componente que lo requiera como dependencia.

Cuando Angular crea una nueva instancia de una clase de componente, determina qué servicios u otras dependencias necesita ese componente al **observar los tipos de parámetros del constructor**:

```
// Angular determina que este componente necesita el servicio 'HeroService'
constructor(private service: HeroService) { }
```

Cuando Angular descubre que un componente depende de un servicio, primero verifica si el inyector tiene alguna instancia existente de ese servicio. Si todavía no existe una instancia de servicio solicitada, el inyector crea una utilizando el proveedor registrado y la agrega al inyector antes de devolver el servicio a Angular de forma que pueda ser reutilizada si otro componente requiere la misma dependencia.

Cuando todos los servicios solicitados se han resuelto y devuelto, Angular puede llamar al constructor del componente con esos servicios como argumentos.

Debe registrar al menos un proveedor de cualquier servicio que vaya a utilizar. El proveedor puede ser parte de los metadatos del servicio, haciendo que ese servicio esté disponible en todas partes, o puede registrar proveedores con módulos o componentes específicos. Es decir, los proveedores se registran en los metadatos de la función decoradora del servicio en `@Injectable ()`, o en los metadatos del módulo `@NgModule()` o del componente `@Component ()`:

Por defecto, cuando creamos un servicio con el comando `ng generate service` de la *'Angular CLI'* registra un proveedor en el inyector raíz para el servicio creado incluyendo metadatos en la función decoradora `@Injectable()`. Cuando se provee un servicio en el nivel raíz o *'root'* el servicio es único y compartido por todos aquellos componentes que lo necesiten. También permite a Angular optimizar la aplicación eliminando los servicios que no se estén utilizando:

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

Cuando se registra un servicio a nivel de módulo, la misma instancia del servicio está disponible para todos los componentes de ese módulo. Esto es, se crea un *singleton* por cada módulo en el que se provea un servicio. Si el mismo servicio se registra en dos módulos, cada módulo recibirá su propia instancia. Para registrar un servicio a nivel de módulo se utiliza la propiedad `providers:[]` de la función decoradora `@NgModule()`:

```
@NgModule({
  providers: [
  BackendService,
  Logger
 ],
 //...
})
```

Cuando se registra un servicio a nivel de componente, se obtiene una nueva instancia del servicio por cada nueva instancia del componente. Para registrar un servicio a nivel de componente se utiliza la propiedad `providers:[]` de la función decoradora `@Component()`:

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ CalculatorService ]
})
```

Para crear un servicio de ejemplo abrimos el terminal/consola y escribimos el comando, lo que creará el servicio en el fichero `calculator.service.ts`:

```
`ng generate service <services/calculator>`  // Substituir 'calculator' por el nombre del servicio
```

Esto genera le servicio en Angular que es una clase decorada con `@Injectable()`:

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

@Injectable({
  providedIn: 'root'
})
export class CalculatorService {
  constructor() { }
}
```

Routing & Navigation

https://angular.io/guide/router
https://angular.io/guide/router#summary
https://academia-binaria.com/paginas-y-rutas-angular-spa/

El enrutador o *'Router'* es el componente en Angular que gestiona la navegación de una vista a otra. Para ello se configura un 'Router' en la aplicación.

`<base href>`

https://angular.io/guide/router#base-href

La mayoría de las aplicaciones de enrutamiento deben agregar un elemento `<base>` al fichero `index.html` como el primer elemento hijo en la etiqueta `<head>` para indicarle al enrutador cómo componer las URL de navegación.

Si la carpeta `app/` es la carpeta raíz de la aplicación, es decir, se ha mantenido la estructura por defecto al crear la aplicación en Angular, esta etiqueta tendrá la siguiente forma:

```
<html>
  <head>
    <base href="/">
    <!-- ... -->
  </head>
  <!-- ... -->
</html>
```

Importar el módulo 'RouterModule'

https://angular.io/guide/router#router-imports

El *'Router'* o enrutador en Angular es un servicio opcional, lo que significa que no forma parte del core de Angular. Está en su propio paquete de biblioteca, el `@angular/router` . Tendremos que importar aquellas partes que se necesiten, tal y como se haría con cualquier otro paquete de Angular:

```
// Por ejemplo importamos 'RouterModule' y 'Routes' de la biblioteca
import { RouterModule, Routes } from '@angular/router';
```

Configuración

https://angular.io/guide/router#configuration

Una aplicación enrutada en Angular tiene una instancia única del servicio de `Router` . Cuando la URL del navegador cambia, ese enrutador busca una ruta correspondiente desde la cual puede determinar el componente a mostrar.

Un enrutador no tiene rutas hasta que se configura. El siguiente ejemplo crea cinco definiciones de ruta, configura el enrutador mediante el método `RouterModule.forRoot()` y agrega el resultado al array de `imports:[]` del módulo raíz `AppModule` en el fichero `app.module.ts` :

```
import { Routes, RouterModule } from '@angular/router';
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only. Muestra información de cada evento de enrutado en la cons
    )
```

```
    // other imports here
  ],
  ...
})
export class AppModule { }
```

El array de `Routes` llamado `appRoutes` describe las rutas de la aplicación, lo que permitirá navegar por ella. Este array se pasa al método `RouterModule.forRoot()` en las importaciones del módulo para configurar el enrutador.

Cada `Route` asigna una ruta URL a un componente. No hay barras diagonales en el *path*. El enrutador analiza y crea la URL final, lo que permite utilizar rutas relativas y absolutas al navegar entre las vistas de la aplicación.

El `:id` en la segunda ruta de ejemplo es un token para un parámetro de ruta. En una URL como por ejemplo "/hero/42", el valor '42' es el valor del parámetro 'id'. El componente *'HeroListComponent'* del ejemplo recibirá ese valor y podrá usarlo para lo que necesite.

La propiedad *'data'* en la tercera ruta es un lugar para almacenar datos arbitrarios asociados con esta ruta específica. Se puede acceder a la propiedad de datos dentro de cada ruta activada. Se puede utilizar para guardar el título de la página o cualquier otro dato estático y *'read-only'*.

El *path* vacío de la cuarta ruta de ejemplo representa el ***path* por defecto** de la aplicación. El lugar al que ir cuando la ruta en la URL está vacía, como sería http://my-application.com suele ser la *home* o página inicial pero puede ser cualquier otra y es aquí donde se define. En el ejemplo la ruta por defecto *'redirecciona'* a la URL "/heroes", que como se indica en la tercera ruta, será gestionada por el componente `HeroLisComponent`. Para hacer redirecciones se utiliza la propiedad `redirectTo:`.

La ruta `**` en la última ruta es un comodín o *'wildcard'*. El enrutador seleccionará esta ruta si la URL solicitada no coincide con ninguna ruta para las rutas definidas anteriormente en la configuración. Esto es útil para mostrar una página de tipo *"404 - Not found"* o redirigir a otra ruta.

El orden de las rutas en la configuración es importante y esto es por diseño. El enrutador utiliza una estrategia *'first-match wins'* empezando de arriba hacia abajo. Cuando coincide la ruta con el patrón de la URL carga el componente. Debido a esta estrategia las rutas más específicas deben colocarse al principio y las genéricas o por defecto más abajo. En la configuración anterior de ejemplo, las rutas con una ruta estática se enumeran primero, seguidas de una ruta vacía, que coincide con la ruta predeterminada. La ruta comodín ocupa el último lugar porque coincide con todas las URL y debe seleccionarse solo si no se encuentra una coincidencia primero.

Router outlet

https://angular.io/guide/router#router-outlet

El `RouterOutlet` es una directiva de la biblioteca de enrutadores que se utiliza como un componente. Actúa como un marcador de posición que marca el lugar en la plantilla donde el enrutador debe mostrar los componentes para esa salida.

La idea general de una **SPA es tener una única página que cargue dinámicamente otras vistas**. Normalmente la página contenedora mantiene el menú de navegación, el pie de página y otras áreas comunes. Y deja un espacio para la carga dinámica. Para ello necesitamos saber **qué componente cargar y dónde mostrarlo**. El componente vendrá indicado por el array de `Routes[]`.

Por ejemplo, si tenemos los siguintes ficheros:

```
// app.module.ts
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
```

```
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

```
<!-- app.component.html -->
<navbar></navbar>
<main class="container">
  <router-outlet></router-outlet>
  <!-- Dynamic content here! -->
</main>
<footer></footer>
```

De esta forma modificamos el fichero `app.component.html` que es la plantilla del componente raíz con la etiqueta `<router-outlet></router-outlet>` de forma que según la URL, que por ejemplo puede ser "/heroes", Angular cargará el componente indicado `HeroListComponent` y substituirá dinámicamente la etiqueta `<router-outlet></router-outlet>` por la vista de ese componente. De esta forma, tenemos una página dinámica que comporte los menús de navegación y el pie de página y con un contenido dinámico que se carga según la URL.

Router links

https://angular.io/guide/router#active-router-links

Una vez definidas las rutas y los componentes asociados sólo queda ver como navegar por la aplicación, ya sea porque el usuario escribe la dirección directamente en la barra de direcciones del navegador (sólo suele suceder la primera vez) o porque acciones del usuario como clicks en botones o en enlaces dentro de la aplicación.

La directiva `RouterLink` en las etiquetas `<a>` le dan al enrutador control sobre esos elementos. Las rutas de navegación son fijas, por lo que puede asignar una cadena de texto al routerLink (un enlace "de una sola vez"):

```
<!-- app.component.html -->
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active important">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

La directiva *'RouterLinkActive'* alterna las clases css para enlaces activos de *'RouterLink'* basados en el *'RouterState'* actual. Esta directiva tiene la forma `routerLinkActive="..."` e indica la clase css entre comillas que se aplicará cuando el enlace esté activo y que se eliminará cuando no esté activo. Si se aplican más de una clase se separan con un espacio en blanco.

Los enlaces de ruta activos caen en cascada a través de cada nivel del árbol de ruta, por lo que los enlaces de enrutador padre e hijo pueden estar activos al mismo tiempo.

RouterState

https://angular.io/guide/router#router-state

Después del final de cada ciclo de vida de navegación exitoso, el enrutador crea un árbol de objetos `ActivatedRoute` que conforman el estado actual del enrutador. Puede acceder al `RouterState` actual desde cualquier lugar de la aplicación utilizando el servicio `Router` y la propiedad `routerState`.

Cada `ActivatedRoute` en el `RouterState` proporciona métodos para recorrer hacia arriba y hacia abajo el árbol de ruta para obtener información de las rutas padre, hijo y hermano.

ActivatedRoute

https://angular.io/guide/router#activated-route

La ruta y los parámetros están disponibles a través de un servicio de enrutador inyectado llamado `ActivatedRoute`. Tiene una gran cantidad de información útil que incluye:

- *'URL'* -> un observable del path o paths de la ruta, representado por un array de strings por cada parte del path.
- *'data'* -> un observable que contiene el objeto 'data' suministrado a la ruta.
- *'paramMap'* -> un observable que contiene un 'map' con los parámetros requeridos y opcionales especificados para la ruta.
- *'outlet'* -> el nombre del 'RouterOutlet' usado para renderizar la ruta.
- *'routeConfig'* -> la configuración usada para la ruta que contiene el path original
- *'parent'* -> el 'ActivatedRoute' padre cuando es una ruta hijo.
- *'firstChild'* -> contiene el primer 'ActivatedRoute' de la lista de rutas hijo.
- *'children'* -> contiene todas las rutas hijo.

```
ngOnInit() {
  let id = this.route.snapshot.paramMap.get('id');

  this.hero$ = this.service.getHero(id);
}
```

Router events

https://angular.io/guide/router#router-events

Durante la navegación, el *'Router'* emite eventos a través de la propiedad `Router.events`. Entre el evento inicial hasta el evento final se producen muchos eventos intermedios, como sería cargar la configuración, reconocer la ruta, etc… Estos eventos se registran en la consola cuando la opción `enableTracing` también está habilitada.

Aquí está la lista de todos los eventos que se producen.

Components & Templates

Displaying Data

https://angular.io/guide/displaying-data

Puede mostrar datos vinculando controles en una plantilla HTML a las propiedades de un componente angular.

La forma más fácil de mostrar una propiedad de un componente es vincular el nombre de la propiedad mediante **interpolación**. Para ello se utiliza el nombre de la propiedad entre doble corchetes ({{}}) en la plantilla HTML:

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

@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
    `
})
```

```
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

Angular extrae automáticamente el valor de las propiedades de `title` y `myHero` del componente e inserta esos valores en el navegador. Angular mantiene actualizados los valores en la vista cuando cambia el valor de estas propiedades.

Nótese que no hemos instanciado de forma explícita el componente `AppComponent`. Angular se encarga de ello. En los metadatos del objeto pasado a la función decoradora `@Component()` le indicamos a Angular la etiqueta `<app-rot>`. Cuando Angular encuentra esa etiqueta en el fichero `index.html` crea una instancia del componente `AppComponent` y lo renderiza dentro de la etiqueta `<app-root>`.

Directiva `*ngFor`

Cuando la propiedad es una array de valores, podemos recorrer la lista para visualizar los valores mediante la directiva `*ngFor`:

```
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  myHero = this.heroes[0];
}
```

```
<h1>{{title}}</h1>
<h2>My favorite hero is: {{myHero}}</h2>
<p>Heroes:</p>
<ul>
  <li *ngFor="let hero of heroes">
    {{ hero }}
  </li>
</ul>
```

Angular duplica la etiqueta `<li>` por cada elemento de la lista y muestra el valor de la variable `hero`.

En este caso la directiva `*ngFor` se emplea para visualizar un array aunque se puede utilizar para repetir items de cualquier objeto iterable.

Visualizar datos de un objeto

Aunque en el ejemplo hemos utilizado un array, en aplicaciones reales se suele iterar sobre objetos que forman parte de la lógica de la aplicación.

Crearemos un clase `Hero` que contendrá las propiedades y utilizaremos la directiva `*ngFor` para recorrer un array de objetos `Hero` y visualizar el nombre:

```
# Crear la clase 'Hero'
ng generate class hero
```

```
// 'hero.ts'
export class Hero {
  constructor( public id: number, public name: string) { }
}
```

```
// 'app.component.ts'
heroes = [
  new Hero(1, 'Windstorm'),
  new Hero(13, 'Bombasto'),
  new Hero(15, 'Magneta'),
  new Hero(20, 'Tornado')
];
myHero = this.heroes[0];
```

```
<!-- app.component.html -->
<h1>{{title}}</h1>
<h2>My favorite hero is: {{myHero.name}}</h2>
<p>Heroes:</p>
<ul>
  <li *ngFor="let hero of heroes">
    {{ hero.name }}
  </li>
</ul>
```

Directiva `*ngIf`

A veces, una aplicación necesita mostrar una vista o una parte de una vista solo en circunstancias específicas. Para ello se utiliza la directiva `*ngIf`, similar al `if` condicional de la mayoría de lenguajes:

```
<p *ngIf="heroes.length > 3">There are many heroes!</p>
```

En el ejemplo, si la expresión es `true` Angular añade la etiqueta `<p>` al DOM y se muestra el mensaje.

Con esta directiva Angular no se limita a mostrar u ocultar la etiqueta o su contenido (es decir, no es un simple `display: none;` en CSS). Lo que hace es añadir o eliminar el elemento y todos sus descendientes del DOM, lo que mejora la eficiencia.

Component Styles

https://angular.io/guide/component-styles#component-styles

En las aplicaciones en Angular se aplica el estilo con CSS estándar a excepción de algunos selectores especiales.

Ámbito de los estilos CSS

Los estilos especificados en los metadatos del `@Component()` sólo se aplican a ese componente. Los estilos **tampoco son heredados** por ningún componente anidado dentro de la plantilla.

Esta modularidad de los estilos ofrece ciertas ventajas:

- Se puede usar los nombres y selectores de clase CSS que tengan más sentido en el contexto de cada componente.
- Los nombres y selectores de clase son locales para el componente y no colisionan con las clases y selectores utilizados en otras partes de la aplicación.
- Los cambios en los estilos en otras partes de la aplicación no afectan los estilos del componente.
- Puede cambiar o eliminar el código CSS del componente sin buscar en toda la aplicación para encontrar dónde más se usa el código.

Selectores especiales

:host

El selector de pseudo-clase `:host` se utiliza para seleccionar estilos en el elemento que aloja el componente. Este selector es la única forma de apuntar al elemento que aloja el componente ya que el elemento `:host` está dentro de la plantilla del componente padre.

```
/* hero-details.component.css */
:host {
  display: block;
  border: 1px solid black;
}
```

Podemos usar la forma funcional para aplicar estilos de forma condicional:

```
/* El estilo se aplica al elemento 'host' pero sólo cuando tiene la clase '.active' */
:host(.active) {
  border-width: 3px;
}
```

:host-context

A veces es útil aplicar estilos basados en alguna condición fuera de la vista de un componente. El selector `:host-context()` busca una clase CSS en cualquier ancestro del elemento host del componente, hasta la raíz del documento. El selector `:host-context()` es útil cuando se combina con otro selector.

```
/* Se aplica el estilo `background-color` a todas las etiquetas '<h2>' dentro del componente solo si algún elemento del
:host-context(.theme-light) h2 {
  background-color: #eef;
}
```

Loading component styles

https://angular.io/guide/component-styles#loading-component-styles

Hay varias formas de añadir estilo a un componente:

- Configurando `styles` o `stylesurls` en los metadatos del componente
- Dentro de la plantilla HTML
- Importando las hojas de estilo con `import`

Estilo en los metados del componente

Podemos añadir los estilos en los metadatos del componente. La forma más habitual es utilizar la propiedad `styleUrls` y cargar un fichero externo que contenga los estilos CSS:

```
// CSS in file
@Component({
  selector: 'app-root',
  template: `
    <h1>Tour of Heroes</h1>
    <app-hero-main [hero]="hero"></app-hero-main>
  `,
  styleUrls: ['./hero-app.component.css']
})
export class HeroAppComponent {
/* . . . */
}
```

Para componentes sencillos podemos incluir los estilos directamente en la propiedad `styles`:

```
// CSS inline
@Component({
  selector: 'app-root',
  template: `
    <h1>Tour of Heroes</h1>
    <app-hero-main [hero]="hero"></app-hero-main>
  `,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
/* . . . */
}
```

Las dos propiedades no son excluyentes, de forma que podemos utilizar ambas en un mismo componente.

Como ya hemos comentado, tanto los estilos en el fichero CSS externo como los estilos aplicados directamente **sólo** aplican al componente donde se definen. No son heredados por otro componente que esté dentro de la plantilla ni son visibles por otro componente del proyecto.

Template inline styles

Podemos incrustar estilos CSS directamente en la plantilla HTML colocándolos dentro de las etiquetas `<style>`:

```
@Component({
  selector: 'app-hero-controls',
  template: `
    <style>
      button {
        background-color: white;
        border: 1px solid #777;
      }
    </style>
    <h3>Controls</h3>
    <button (click)="activate()">Activate</button>
  `
})
```

Template link tags

También podemos escribir etiquetas `<link>` en la plantilla HTML del componente para hacer referencia a un fichero CSS. En este caso habrá que incluir el fichero en la carpeta `assets` de forma que la CLI lo incluya al compilar el proyecto:

```
@Component({
  selector: 'app-hero-team',
  template: `
    <!-- We must use a relative URL so that the AOT compiler can find the stylesheet -->
    <link rel="stylesheet" href="../assets/hero-team.component.css">
    <h3>Team</h3>
    <ul>
      <li *ngFor="let member of hero.team">
        {{member}}
      </li>
    </ul>`
})
```

CSS @imports

También podemos importar archivos CSS en los archivos CSS utilizando la regla estándar CSS `@import`. En este caso, la URL es relativa al archivo CSS desde el que se está importando.

```
/* The AOT compiler needs the `./` to show that this is local */
@import './hero-details-box.css';
```

External and global style files

Al compilar con la CLI, debemos configurar el fichero `angular.json` para incluir todos los assets externos, incluidos los archivos de estilo externos.

Los archivos de estilo globales se registran en la propiedad `styles` que de forma predeterminada está configurada con el archivo global `styles.css`:

```
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": {
      "styles": [
        "src/styles.css",
        "src/more-styles.css",
      ],
       // ...
    }
  }
}
```

Non-CSS style files

Si estamos utilizando la herramienta CLI para compilar y construir el proyecto, podemos usar preprocesadores como **SASS**, **LESS** o **STYLUS**. Podemos especificar el fichero en la propiedad `@Component.styleUrls` con la extensión apropiada:

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

Al compilar el proyecto con la herramienta CLI, ejecutará el preprocesador correspondiente.

Sólo podemos usar preprocesadores en ficheros externos. Los estilos en la propiedad `@Component.styles` tiene que ser escritos en CSS.

User Input

https://angular.io/guide/user-input

Las acciones del usuario como hacer clic en un enlace, presionar un botón e ingresar texto generan eventos DOM.

Binding to user input events

Puede usar Angular event bindings para responder a cualquier evento DOM.

Muchos eventos DOM se desencadenan por un input del usuario. La vinculación a estos eventos proporciona una forma de obtener información del usuario.

Para enlazar un evento DOM, encierre el nombre del evento DOM entre paréntesis y asígnele una declaración de plantilla entrecomillada:

```
<button (click)="onClickMe()">Click me!</button>
```

`(click)` identifica el evento 'click' del botón como el objetivo del enlace. `"onClickMe()"` es el método del componente que será invocado cuando se produzca el evento 'click' sobre el botón.

Get user input from the $event object

Los eventos DOM llevan una carga útil de información que puede ser útil para el componente. Esta información se pasa al componente a través de la variable `$event` :

```
@Component({
  selector: 'app-click-me',
  template: `
    <input (keyup)="onKey($event)">
    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

Las propiedades del objeto `$event` varían dependiendo del tipo de evento DOM. Por ejemplo un evento del ratón incluye información diferente de eventos producidos por pulsaciones de teclas.

Todos los objetos de eventos DOM estándar tienen la propiedad *'target'*, que es una referencia al objeto que lanzó el evento.

En el ejemplo *'target'* se refiere al elemento `<input>` y `event.target.value` devuelve el contenido actual de ese elemento. Mediante la interpolación mostramos el valor de la variable en `{{values}}` .

El problema de usar esta forma es que en el ejemplo se indica `any` como el tipo de la variable `$event` , lo que simplifica el código pero no tenemos información que nos revele las propiedades del objeto. Podemos indicar el tipo, como por ejemplo `onKey(event: KeyboardEvent) { }` , lo que significa que ahora el componente tiene demasiada información de la plantilla lo que rompe la separación de responsabilidades.

Get user input from a template reference variable

https://angular.io/guide/user-input#get-user-input-from-a-template-reference-variable
https://angular.io/guide/template-syntax#template-reference-variables-var

Para mantener la separación de responsabilidades podemos usar *variables de referencia*. Estas variables proveen un acceso directo al elemento dentro de la plantilla que lanza el evento DOM. Para declarar una variable de referencia usamos la almohadilla (`#`):

```
@Component({
  selector: 'app-key-up2',
```

```
  template: `
    <input #box (keyup)="onKey(box.value)">
    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v2 {
  values = '';
  onKey(value: string) {
    this.values += value + ' | ';
  }
}
```

La variable de referencia `#box` declarada en el elemento `<input>` se refiere al elemento en sí mismo. De esta forma le pasamos directamente el valor al componente. Ya no requiere conocimiento de `$event` y su estructura.

Key event filtering (with 'key.event')

El manejador de evento `(keyup)` se activa por una pulsación de cualquier tecla. Podemos filtrar y reducir el 'ruido' si se lanza el evento cuando se pulse una tecla concreta. En el ejemplo sólo se lanza el evento cuando se pulse la tecla 'ENTER' al utilizar el pseudoevento `keyup.enter`:

```
@Component({
  selector: 'app-key-up3',
  template: `
    <input #box (keyup.enter)="onEnter(box.value)">
    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v3 {
  value = '';
  onEnter(value: string) { this.value = value; }
}
```

Sintaxis de las plantillas

https://angular.io/guide/template-syntax#template-syntax

Las aplicaciones en Angular gestionan lo que el usuario ve y puede hacer, logrando esto mediante la interacción de una instancia de la clase `Component` (el componente) y su plantilla orientada al usuario.

Siguiendo los patrones *Model-View-Controller (MVC)* o *Model-View-ViewModel (MVVM)*, en Angular el componente hace el papel de controlador/viewModel y la plantilla hace el papel de vista.

HTML en las plantillas

https://angular.io/guide/template-syntax#html-in-templates

HTML es el lenguaje utilizado en las plantillas de Angular. Casi toda la sintaxis HTML es una sintaxis de plantilla válida. La etiqueta `<script>` es una notable excepción. Evitando su uso se evita problemas de seguridad y el riesgo de ataques por inyección de comandos.

Otras etiquetas que no tienen sentido en Angular son `<html>`, `<body>` o `<base>`.

Interpolation and Template Expressions

https://angular.io/guide/template-syntax#interpolation-and-template-expressions

La interpolación le permite incorporar cadenas calculadas en el texto entre etiquetas de elementos HTML y dentro de las asignaciones de atributos. Las expresiones de plantilla son lo que se usa para calcular esas cadenas.

Interpolation {{…}}

https://angular.io/guide/template-syntax#interpolation-

La interpolación se refiere a la capacidad de poder incrustar expresiones dentro de etiquetas HTML. Por defecto, la interpolación utiliza como delimitador las llaves dobles, *{{ y }}*.

```
<h3>Current customer: {{ currentCustomer }}</h3>
```

El texto entre llaves suele ser el nombre de una propiedad del componente. Angular reemplaza ese nombre con el valor de la propiedad del componente correspondiente.

```
<p>{{title}}</p>
<div><img src="{{itemImageUrl}}"></div>
```

En términos más generales, el texto entre llaves es una expresión de plantilla que Angular primero evalúa y luego convierte en una cadena:

```
<p>The sum of 1 + 1 is {{1 + 1}}.</p>
```

La expresión puede invocar métodos del componente host como `getVal()`:

```
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}.</p>
```

Template Expressions (Expresiones de plantilla)

https://angular.io/guide/template-syntax#template-expressions

Una expresión de plantilla produce un valor y normalmente aparece dentro de llaves dobles, `{{}}`. Angular ejecuta la expresión y la asigna a una propiedad de un elemnto; el elemento podría ser un elemento HTML, un componente o una directiva.

En una interpolación como `{{1 + 1}}` las llaves rodean una expresión aunque no siempre se utilizan las llaves dobles. En un *'property binding'*, la expresión de plantilla aparece entre comillas y tiene la forma `[property]="expression"`.

En términos de sintaxis, las expresiones de plantilla son muy parecidas a Javascript. Muchas expresiones en Javascript son expresiones de plantilla legales con algunas excepciones:

- Asignaciones (=, +=, -=, …)
- Operadores como `new`, `typeof`, `instanceof`, …
- *Chaining expresions* con ; o ,
- Los operadores de incremento o decremento (++ y --)
- Algunos operadores de ES2015
- No hay soporte para operadores a nivel de bits como | y &.

El contexto de la expresión normalmente es el **componente**, de forma que en la expresión se hace referencia a propiedades del componente. En el ejemplo 'recommended' y 'itemImageUrl2' se refieren a propiedades del componente:

```
<h4>{{recommended}}</h4>
<img [src]="itemImageUrl2">
```

El contexto de la expresión también puede ser una variable dentro del template, como por ejemplo hacer referencia a un `input`:

```
<label>Type something:
  <input #customerInput>{{customerInput.value}}
</label>
```

Por último el contexto de la expresión puede ser tanto una variable dentro de la plantilla como una variable dentro de una directiva `*ngFor`:

```
<ul>
  <li *ngFor="let customer of customers">{{customer.name}}</li>
</ul>
```

Template statements (Declaraciones de plantilla)

https://angular.io/guide/template-syntax#template-statements

Un *template statements* responde a un evento lanzado por un elemento, componente o directiva y tienen la forma `(event)="statement"`.

Al igual que en las expresiones, normalmente el contexto de una declaración es una instancia del componente:

```
<button (click)="deleteHero()">Delete hero</button>
```

En el ejemplo anterior `(click)="deleteHero()"` la palabra 'deleteHero' es un método del componente.

El contexto también puede hacer referencia a propiedades del propio contexto de la plantilla, variables que hacen referencia a elementos de la plantilla (`#heroForm`) o variables creadas por una directiva como `*ngFor`:

```
<button (click)="onSave($event)">Save</button>
<button *ngFor="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}</button>
<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>
```

Data-Binding

https://angular.io/guide/template-syntax#binding-syntax-an-overview

Data-binding es un mecanismo para coordinar lo que ven los usuarios, específicamente con los valores de datos de la aplicación. Si bien se pueden extraer e insertar valores dentro del HTML, una aplicación es más fácil de escribir, leer y mantener si un framework específico se encarga de estas tareas.

Angular proporciona varios tipos de *data-binding* que se pueden agrupar según la dirección del flujo:

	Markup	**Nombre**
DOM <------- Component	{{expression}}	*'interpolation'*

	Markup	Nombre
DOM <------- Component	[target] = "expression"	*'property binding'*
DOM <------- Component	bind-target = "expression"	
----------	----------	----------
DOM -------> Component	(target) = "statement"	*'event binding'*
DOM -------> Component	on-target = "statement"	
----------	----------	----------
DOM <------> Component	[(target)] = "expression"	*'two-way binding'*
DOM <------> Component	bindon-target = "expression"	*'two-way binding'*

El *data-binding* trabaja con las **propiedades de los elementos del DOM, componentes y directivas** y no con los atributos HTML.

La diferencia entre una propiedad DOM y un atributo HTML es que los atributos están definidos por el HTML y las propiedades son accesibles por el DOM. Una regla para distinguir un atributo de una propiedad es que los atributos inicializan las propiedades del DOM y luego terminan. **Los valores de las propiedades pueden cambiar mientras que el valor del atributo no cambia**.

El atributo HTML y la propiedad DOM no son lo mismo, incluso cuando tienen el mismo nombre.

```
<input type="text" value="Sarah">
```

Cuando el navegador renderiza esta etiqueta, crea el correspondiente DOM con la propiedad `value` inicializada con "Sarah". Cuando el usuario introduce la palabra "Sally" en el `<input>` la propiedad `value` cambia a "Sally".

Sin embargo, si miramos el atributo HTML `value` con la función `input.getAttribute('value')` vemos que su valor sigue siendo "Sarah".

En definitiva, en Angular el único papel de los atributos es inicializar el elemento y el estado de la directiva. Cuando se crea un *data-binding* se trata exclusivamente de propiedades y eventos del objeto de destino.

Binding targets

https://angular.io/guide/template-syntax#binding-targets

El objetivo de un enlace de datos es algo en el DOM. Depende del tipo, el objetivo puede ser una propiedad (de un elemento, componente o directiva), un evento (de un elemento, componente o directiva) y de forma excepcional con algunos atributos:

- **Evento** (elemento, componente o directiva)

```
<button (click)="onSave()">Save</button>
<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>
<div (myClick)="clicked=$event" clickable>click me</div>
```

- **Two-way** (eventos y propiedades)

```
<input [(ngModel)]="name">
```

- **Atributos** (de forma excepcional)

```
<button [attr.aria-label]="help">help</button>
```

- **Clases**

```
<div [class.special]="isSpecial">Special</div>
```

- **Style**

```
<button [style.color]="isSpecial ? 'red' : 'green'">
```

Property binding `[property]`

https://angular.io/guide/template-syntax#property-binding-property

El enlace de propiedades se utiliza para establecer propiedades de elementos de destino o decoradores de directivas `@Input()`. La propiedad del elemento de destino se encierra entre llaves `[]` mientras que la expresión de plantilla se encierra entre comillas y hace referencia a la propiedad del componente:

```
<img [src]="heroImageUrl">
<app-hero-detail [hero]="currentHero"></app-hero-detail>
<div [ngClass]="{'special': isSpecial}"></div>
```

One-way in

El enlace de propiedad fluye un valor en una dirección, desde la propiedad de un componente a una propiedad del elemento de destino.

El enlace de propiedades no puede usarse para leer o extraer valores del elemento de destino. Del mismo modo, el enlace de propiedades no puede utilizarse para llamar a un método en el elemento de destino. Para eso se utiliza el 'enlace de eventos'.

Este es un ejemplo del enlace de propiedad más común en que una propiedad de un elemento se establece con el valor de la propiedad del componente:

```
<!-- Se enlaza la propiedad 'src' del elemento '<img>' con la propiedad 'itemImageUrl' del componente -->
<img [src]="itemImageUrl">
```

Otros ejemplos:

```
<!-- Enlazamos el estado 'disabled' del botón con la propiedad 'isUnchanged'-->
<button [disabled]="isUnchanged">Disabled Button</button>

<!-- También podemos establecer el valor de la directiva 'ngClass' -->
<p [ngClass]="classes">[ngClass] binding to the classes property making this blue</p>
```

Property binding vs interpolation

A menudo se puede elegir entre interpolación y enlace de propiedad. Los siguientes ejemplos son equivalentes:

```
<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i> image.</p>
<p><img [src]="itemImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{interpolationTitle}}" is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="propertyTitle"></span>" is the <i>property bound</i> title.</p>
```

La interpolación es la forma más conveniente en muchos casos. Cuando se trata de 'strings' no hay ninguna razón que obliga a usar una forma u otra. En cambio, cuando se establece la propiedad de un elemento con yun valor que no sea una cadena **hay que usar un enlace de propiedad**.

Content security

Por seguridad, Angular no permite que se utilicen etiquetas HTML con `<script>` ni en interpolación ni con enlace de propiedad.

Attribute, class and style bindings

https://angular.io/guide/template-syntax#attribute-class-and-style-bindings

(TODO)

Enlaces de interés

- https://angular.io/docs
- https://angular.io/guide/architecture
- https://academia-binaria.com/cursos/angular-basic

Licencia