Version : 6.0.0 (7/5/18)

Introducción

Angular es un framework JavaScript, gratuito y Open Source, creado por Google y destinado a facilitar la creación de aplicaciones web modernas de tipo SPA (Single Page Application).

Algunos incluso llegan a decir de Angular que es lo que debería haber sido HTML si se hubiese pensado desde el principio para crear aplicaciones web, y no documentos. Casi nada.

Su primera versión, AngularJS, se convirtió en muy poco tiempo en el estándar de facto para el desarrollo de aplicaciones web avanzadas.

En septiembre de 2016 Google lanzó la versión definitiva de lo que llamó en su momento Angular 2, y que ahora es simplemente Angular. Este nuevo framework se ha construido sobre años de trabajo y feedback de los usuarios usando AngularJS (la versión 1.x del framework). Su desarrollo llevó más de 2 años. No se trata de una nueva versión o una evolución de AngularJS, sino que es un nuevo producto, con sus propios conceptos y técnicas. Además, Angular utiliza como lenguaje de programación principal TypeScript, un súper-conjunto de JavaScript/ECMAScript que facilita mucho el desarrollo.

Dado que Angular es un framework, ofrece muchas más "opiniones" y funcionalidades de serie que una simple biblioteca. Con otro software similar, lo más común es tener que usar varias bibliotecas de terceros a la hora de desarrollar una app. Lo más probable es que necesites algunas adicionales para hacer el routing, para la gestión de dependencias, para realizar llamadas a APIs REST, para hacer el testing, etc... También hay muchas decisiones que tomar sobre cómo organizar el código, la arquitectura de la aplicación... Tantas que pueden llegar a abrumar.

Angular ofrece más "opiniones" de serie, ayudándote a arrancar sin intimidarte por la toma de decisiones. Es decir, con Angular ya sabes desde el primer momento cómo organizar el código, cómo se realizan las diferentes tareas que necesitas, la arquitectura de la aplicación, etc.

Una cuestión muy importante es que esta consistencia que te impone también ayuda a las empresas a hacer nuevas contrataciones de programadores o a incorporar a nueva gente a los equipos. Un nuevo programador que retome un proyecto de Angular se siente como en casa de forma rápida, ya que si conoce Angular conoce la manera en la que se hacen las cosas con este framework. Esto facilita también el intercambio de programadores entre proyectos.

Aunque se puede programar en ECMAScript "puro", el equipo de Angular decidió que haría todo el desarrollo con el lenguaje TypeScript, y casi toda la documentación y los ejemplos que encuentras por ahí utilizan este lenguaje.

A mucha gente esto le parecerá un error, pero el criterio que ha seguido el equipo de Angular sobre la variedad de JavaScript a utilizar tiene muchas ventajas.

Una de las primeras es la consistencia en la documentación. Si navegas por la Web intentando encontrar ejemplos y tutoriales de otras bibliotecas de JavaScript vas a ver de todo, pero la única constante es la inconsistencia que existe. Con TypeScript esto no pasa, y toda la sintaxis y la manera de hacer las cosas en el código es la misma, lo que añade coherencia a la información y a la forma de leer el código.

Un componente en Angular es una porción de código que es posible reutilizar en otros proyectos de Angular sin mucho esfuerzo, lo que permite un desarrollo de aplicaciones mucho más ágil, pasando de un "costoso" MVC a un juego de puzles con nuestros componentes.

El diseño de Angular adopta el estándar de los componentes web. Se trata de un conjunto de APIs que te permiten crear nuevas etiquetas HTML personalizadas, reutilizables y auto-contenidas, que luego puedes utilizar en otras páginas y aplicaciones web. Estos componentes personalizados funcionarán en navegadores modernos y con cualquier biblioteca o framework de JavaScript que funcione con HTML.

El soporte directo actual de este estándar por parte de los navegadores es muy reducido (básicamente Chrome a la hora de escribir esto), aunque existen polyfills para suplir esta carencia.

No te dejes engañar por los cambios bruscos de versión en Angular: no significa que haya grandes modificaciones, solo que hay alguna cosa que rompe la compatibilidad (aunque sea mínimamente). Pero puede ser algo muy pequeño y además ofrecerán herramientas automáticas para hacer la migración.

Con Angular la idea es que podemos apostar por este framework a largo plazo. Y esto es algo de suma importancia, sobre todo en proyectos grandes y en empresas de producto, donde los desarrollos se mantienen durante varios años.

Cuando programas raramente vas a hacerlo en un editor de texto plano. Usarás editores avanzados, IDEs y otras herramientas relacionadas.

Las plantillas de Angular almacenan por separado el código de la Interfaz de usuario y el de la lógica de negocio, por lo que puedes sacarle partido a las muchas herramientas ya existentes para editar este tipo de archivos. Otros frameworks como React, por ejemplo, mezclan en un mismo archivo todo el código. Si bien esto puede tener sus ventajas, dificulta el uso de herramientas estándar de desarrollo.

Además, gracias a la popularidad de Angular, los principales editores e IDEs ofrecen ya extensiones para poder trabajar con este framework de la manera más cómoda posible.

Tour de Héroes

Paso 1

Instalación

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

Este comando instala el “ayudante” de angular en nuestra computadora de forma global.

Creación de proyecto

```
ng new tour-de-heroes
```

Este comando crea la carpeta tour-de-heroes e inicializa un proyecto de angular con los archivos y scripts básicos requeridos. Este comando también inicializa un repositorio git para nuestro proyecto.

Iniciar aplicación

```
cd tour-de-heroes
ng serve --open
```

Con el primer comando ingresamos a la carpeta recién creada. El segundo comando inicia el servidor de desarrollo instalado con angular y la opción --open abre el navegador para que veamos el proyecto. Este comando también se encarga de compilar nuestros archivos y queda a la espera de cambios para volver a compilar el proyecto cuando sea necesario.

Primeros pasos

Dentro de la carpeta src encontraremos una carpeta llamada app donde estan los archivos que componen el proyecto actualmente.

app.component.ts contiene el codigo de la clase del componente
app.component.html contiene el template de nuestro componente
app.component.css contiene los estilos privados de nuestro componente.

Abrir el archivo app.component.ts y cambiar la propiedad title para reflejar el nombre del proyecto: Tour de Heroes

```
title = 'Tour de Heroes';
```

Abrir el archivo app.component.html y reemplazar todo su contenido por el siguiente

```
<h1>{{title}}</h1>
```

La doble llave es la sintaxis que utiliza angular para reemplazar el contenido por la propiedad que nosotros definimos.

Dentro de la carpeta src se encuentra el archivo styles.css que contendrá los estilos globales de nuestro proyecto. Dentro de este archivo ponemos el siguiente contenido:

```
/* Application-wide Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[text], button {
  color: #888;
  font-family: Cambria, Georgia;
}
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

A medida que vamos haciendo todos estos cambios vamos a ir viendo como el navegador se refresca de forma automática.

Para detener el proceso de compilación y el servidor de desarrollo se utiliza la combinación de teclas CTRL-C.

Paso 2

Creación de un componente

```
ng generate component heroes
```

Este comando crea el código necesario para un nuevo componente llamado heroes. Esto creará la carpeta y archivos necesarios para este nuevo componente (src/app/heroes).

Si abrimos el archivo heroes.component.ts vemos que contiene 3 partes importantes:

selector: es el nombre con el que usaremos el componente. Se utiliza como un tag HTML.
templateUrl: la ubicación del template de este componente. El inicio "./" indica que se trata del directorio actual.
stylesUrls: es un array que contiene la ubicación de los estilos privados del componente.

Hasta acá es muy similar al componente app que se creó de forma automática al inicializar el proyecto.

Mostrar nuestro componente
En el archivo heroes.component.ts, antes del constructor, definimos una propiedad llamada "heroe" y le asignamos el valor Superman

```
heroe = 'Superman';
```

Reemplazamos el contenido del archivo heroes.component.html por lo siguiente:

```
{{heroe}}
```

Por último incluimos nuestro componente en el archivo app.component.html debajo del título utilizando el tag:

```
<app-heroes></app-heroes>
```

En el navegador se debería ver la palabra Superman debajo del título.

Vamos a una clase en Typescript para definir la estructura de datos de nuestros héroes. Para esto creamos el archivo heroe.ts en la carpeta src/app y escribimos lo siguiente:

```
export class Heroe {
  id: number;
  name: string;
}
```

En este archivo estamos definiendo que nuestros héroes van a tener una propiedad id del tipo number y una propiedad name del tipo string. Todos los héroes que creemos deberán respetar esta estructura.

Para utilizar esta nueva clase, nuestro archivo heroes.component.ts deberá verse así:

```
import { Component, OnInit } from '@angular/core';
import { Heroe } from '../heroe';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  heroe: Heroe = {
    id: 1,
    name: 'Windstorm'
  };

  constructor() { }

  ngOnInit() {
  }

}
```

De esta forma importamos la clase que creamos y la utilizamos como una propiedad más de la clase HeroesComponent.

Por último, modificamos el template del componente para reflejar la nueva estructura de datos.

```
<h2>Detalles de {{ heroe.name }}</h2>
<div><span>id: </span>{{heroe.id}}</div>
<div><span>nombre: </span>{{heroe.name}}</div>
```

Uso de modificadores

Si quisiéramos podríamos hacer uso de los modificadores que nos ofrece Angular, como por ejemplo uppercase, que sirve para convertir un texto a mayúsculas.

Lo podemos implementar de la siguiente forma

```
<h2>Detalles de {{ heroe.name | uppercase }}</h2>
```

Asi estaremos mostrando el nombre de nuestro héroe en mayusculas. Existen varios modificadores más definidos en Angular y los podemos encontrar en https://angular.io/guide/pipes, así como también podemos definir los nuestros.

Edicion de datos y enlace bidireccional (two-way databinding)

Vamos a permitir la edición de los datos, para esto vamos a hacer una modificación en el parte del template del componente que muestra el nombre del héroe. El div que lo contiene deberá quedar así:

```
<div>
    <label>nombre:
      <input [(ngModel)]="heroe.name" placeholder="nombre">
    </label>
</div>
```

El atributo **[(ngModel)]** se utiliza para crear una relación bidireccional entre los datos y el input HTML. Esto quiere decir que si cambiamos el contenido del input se cambiará el dato original, y a la inversa, si por algún agente externo se modifica el dato, el valor del input reflejará el nuevo valor.

Para terminar de poner esto en funcionamiento es necesario que importemos el módulo de formularios de Angular, de lo contrario veremos un error en la consola indicando que ngModel no es una propiedad conocida de input. Para hacer uso de este módulo deberemos incluirlo en el archivo app.module.ts de la siguiente forma:

Primero lo importamos en el inicio del archivo:

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

Y luego lo declaramos en los imports más abajo:

```
imports: [
  BrowserModule,
  FormsModule
],
```

Ahora si, si probamos devuelta la aplicación en el navegador vamos a ver el input con el nombre del héroe y si lo editamos vamos a ver el cambio en el título, incluso respetando el modificador de mayúsculas si es que lo usamos.

Mostrando listas

Vamos a mostrar una lista de heroes, para eso, de momento, necesitamos cargarlos en un archivo. Creamos un archivo llamado mock-heroes.ts en la carpeta src/app y escribimos el siguiente código:

```
import { Heroe } from './heroe';

export const HEROES: Heroe[] = [
  { id: 11, name: 'Superman' },
  { id: 12, name: 'Batman' },
  { id: 13, name: 'Flash' },
  { id: 14, name: 'Capitán América' },
  { id: 15, name: 'Iron Man' },
  { id: 16, name: 'Linterna Verde' },
  { id: 17, name: 'Aquaman' }
];
```

Aca importamos nuestra clase Heroe y la utilizamos para crear un array de objetos Heroe. Ahora solo hace falta importar nuestra constante HEROES en el componente. Para eso escribimos:

```
import { HEROES } from '../mock-heroes';
```

Luego creamos una propiedad llamada heroes dentro de nuestra class HeroesComponent y le asignamos el valor de la constante que acabamos de importar de esta forma:

```
heroes =  HEROES;
```

Para listarlos vamos a agregar la siguiente sección al template de nuestro componente:

```
<h2>Heroes</h2>
<ul class="heroes">
  <li *ngFor="let heroe of heroes">{{heroe.name}}</li>
</ul>
```

El atributo *ngFor es propio de Angular y le indica al framework que debe repetir el elemento en el que fue incluido. Vemos que como contenido tiene “let heroe of heroes”. Esto significa que en cada iteración del bucle for va a extraer un elemento del array héroes a la variable local heroe, mediante la cual podemos acceder a las propiedades de dicho objeto. En nuestro caso el id y el nombre.

Seleccionando elementos

Vamos a modificar el elemento li que teníamos para que quede de la siguiente forma:

```
<li *ngFor="let heroe of heroes" (click)="seleccionarHeroe(heroe)"
[class.activo]="heroe === heroeSeleccionado">{{heroe.name}}</li>
```

La sintaxis (click) nos permite asociar el evento click a un método de la clase de nuestro componente (HeroesComponent). En este caso estamos llamando al método “seleccionarHeroe” y le pasamos como parametro la variable “heroe” del for que creamos antes.

También vemos la sintaxis [class.activo]="heroe === heroeSeleccionado". Esto quiere decir que se aplicará al li la clase “activo” cuando la variable “heroe” (del bucle for) sea igual a la variable heroeSeleccionado de nuestra clase HeroesComponent.

A continuación, modificamos el resto del archivo heroes.component.html para mostrar los detalles del heroe seleccionado:

```
<div *ngIf="heroeSeleccionado">
  <h2>Detalles de {{ heroeSeleccionado.name | uppercase }}</h2>
  <div><span>id: </span>{{heroeSeleccionado.id}}</div>
  <div>
    <label>nombre:
      <input [(ngModel)]="heroeSeleccionado.name" placeholder="nombre">
    </label>
  </div>
</div>
```

*ngIf va a mostrar solo si la variable heroeSeleccionado existe. Después, dentro de este div mostraremos los detalles del heroe seleccionado.

Por último, la clase de nuestro componente (heroes.component.ts) deberá verse así:

```
import { Component, OnInit } from '@angular/core';
import { Heroe } from '../heroe';
import { HEROES } from '../mock-heroes';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {

  heroes = HEROES;
  heroeSeleccionado:Heroe;

  constructor() { }

  ngOnInit() {
  }

  seleccionarHeroe(heroe:Heroe):void {
    this.heroeSeleccionado = heroe;
  }

}
```

Aquí estamos definiendo la variable heroeSeleccionado para que sea del tipo Heroe. Al no asignarle un valor inicialmente, el div que definimos antes no se mostrará.

También definimos el método seleccionarHeroe que se llama en el evento click de los li y recibe como parámetro una variable llamada heroe del tipo Heroe. Este método setea la propiedad heroeSeleccionado al valor que reciba en su parámetro heroe.