# Setup

Para desarrollar una aplicación usando Angular en un sistema local, necesitamos configurar un entorno de desarrollo que incluya la instalación de:

* Node.js (min version required 10.13.0) and npm (min version required 6.11.0)
* Angular CLI
* Visual Studio Code

Instalar Node.js and Visual Studio Code desde sus páginas oficiales.

Pasos para instalar Angular CLI

* Angular CLI puede ser instalado usando node package manager con el comando siguiente: *npm install -g @angular/cli*
* Probar la correcta instalación con el siguiente comando: *ng v*

Angular CLI es una herramienta de interfaz de línea de comandos para construir aplicaciones Angular. Hace que el desarrollo de aplicaciones sea más rápido y fácil de mantener.

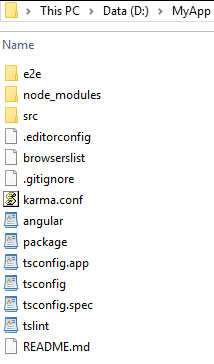
Con CLI, podemos crear proyectos, agregarle archivos y realizar tareas de desarrollo como pruebas, agrupación e implementación de aplicaciones.

* npm install -g @angular/cli: Installs Angular CLI globally
* ng new <project name>: Creates a new Angular application
* ng serve --open: Builds and runs the application on lite-server and launches a browser
* ng generate <schematic> <name>: Creates class, component, directive, interface, module, pipe and service
* ng build: Builds the application

## Creando una aplicación:

1. Cree una aplicación con el nombre 'MyApp' usando el siguiente comando cli: *ng new MyApp*
2. El comando anterior mostrará dos preguntas. La primera pregunta es para crear un archivo routing module (app-routing.module.ts) que se explicará más adelante.
3. La siguiente pregunta es para seleccionar la hoja de estilo a usar en la aplicación. Seleccione CSS.

Podremos ver un ejemplo de la estructura de la aplicación en la imagen projectStructure.



* e2e /: esta carpeta contiene todas las pruebas End to End (e2e) de la aplicación escrita en Jasmine y ejecutada por el Protractor.
* node\_modules /: Node.js crea esta carpeta y coloca todos los módulos npm instalados como se enumeran en package.json.
* src /: todos los archivos relacionados con la aplicación se almacenarán en su interior.
* angular.json: archivo de configuración para Angular CLI donde establecemos varios valores predeterminados y también configuramos qué archivos se incluirán durante la compilación del proyecto.
* package.json: este es un archivo de configuración de nodo que contiene todas las dependencias necesarias para Angular.
* tsconfig.json: este es el archivo de configuración de Typescript donde podemos configurar las opciones del compilador.
* tslint.json: este archivo contiene reglas de linting preferidas por la guía de estilo angular.

Una vez se tiene la aplicación se debe usar "*npm install*" para instalar node\_modules.

Luego para ejecutar la aplicación se utiliza: *ng serve --open*.

# Components & Modules

## Concepts

¿Por qué componentes en angular?

Un componente es el componente básico de una aplicación Angular.

Los componentes enfatizan la separación de problemas y cada parte de la aplicación angular puede escribirse independientemente una de la otra

Los componentes son reutilizables.

Por ejemplo, observe en la aplicación mCart que el componente superior es el componente mCart (AppComponent) que consta de componentes secundarios llamados componente de bienvenida, componente de inicio de sesión, etc.

Revisar:

* app.component.ts
* app.component.html
* app.component.css

Ya hemos visto cómo crear un componente, ahora exploremos los módulos.

* Los módulos en Angular se utilizan para organizar la aplicación. Las aplicaciones Angular son una colección de módulos.
* Un módulo en Angular es una clase con el decorator @NgModule agregado. Los metadatos de @NgModule contendrán las declaraciones de componentes, pipes, directivas, servicios que se utilizarán en toda la aplicación.
* Cada aplicación Angular debe tener un módulo raíz que se carga primero para iniciar la aplicación.
* También podemos tener submódulos que deben configurarse en el módulo raíz.

### Root Module

Revisar:

* app.module.ts localizado dentro del folder app.
* main.ts localizado dentro del folder src.

### Cargando el root component en la página HTML.

Revisar:

* index.html bajo el folder src.

Para ejecutar la aplicación: *ng serve --open*

Para ejecutar la aplicación con otro puerto: *ng serve --open --port 3000*

### Creando un component:

***ng generate component hello***: Este comando generará un folder dentro de app con un archivo ts, html y css.

* En app.module.ts debe agrearse el archivo ts nuevo en bootstrap si se desea colocar como raíz.
* En main.ts debe cambiarse la referencia en body al nuevo html (<app-hello>).

## Template

El lenguaje predeterminado para templates en Angular es HTML. Los templates representan la vista cuyo objetivo es mostrar data y modificarla cuando sea que un evento ocurra.

¿Por qué templates?

Los templates nos permiten separar la capa de visualización del framework. Podremos cambiar esta vista sin romper la aplicación.

Podemos definir un template en dos formas:

* Inline template: Dentro del mismo de la misma clase componente. Dentro del decorator @Component, en el property template (en lugar de templateUrl) podremos agregar el HTML.

@Component({

  selector: 'app-root',

  template: `

        <h1> Welcome </h1>

        <h2> Course Name: {{ courseName }}</h2>

    `,

  styleUrls: ['./app.component.css']

})

* External template: es el predeterminado en Angular

### Elementos de Template

* HTML
* Interpolation: La interpolación es una de las formas de enlace de datos (data binding) donde podemos acceder a los datos del componente en un template. Para la interpolación, utilizamos llaves dobles {{}}.
* Template Expressions: El texto dentro de {{}} es llamado template expresion: {{ expression }}. Angular primero evalúa la expresión y devuelve el resultado como una cadena. El alcance de un template expression es una instancia de componente. Eso significa que, si escribimos {{ courseName }}, courseName debería ser propiedad del componente al que está vinculada este template.
* Template Statements: son las declaraciones que responden a un evento de usuario:

(event) = statement

Por ejemplo (click) = "changeName()"

Esto es llamado event binding. En Angular todos los eventos deben ir en ().

Ejemplo en hello.component.

## Change Detection

Ahora exploremos cómo Angular detecta los cambios y los actualiza en la aplicación en los lugares respectivos. Las aplicaciones Angular se ejecutan más rápido que las aplicaciones Angular 1.x debido a su mecanismo mejorado de detección de cambios.

Entonces, ¿qué es el mecanismo de detección de cambios y cómo ayuda a ejecutar aplicaciones angulares tan rápido?

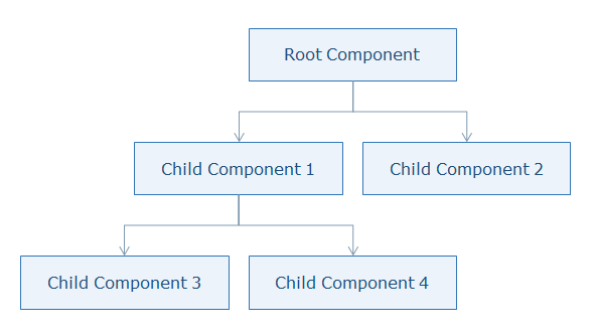
* La detección de cambios es un proceso en Angular que mantiene las vistas sincronizadas con los modelos.
* En Angular, el flujo es unidireccional de arriba hacia abajo en un árbol de componentes. Un cambio en una aplicación web puede ser causado por eventos, llamadas Ajax y temporizadores que son asíncronos.

¿Quién informa a Angular sobre los cambios?

* Zones informa a Angular sobre los cambios en la aplicación. Detecta automáticamente todas las acciones asincrónicas en tiempo de ejecución en la aplicación.

Ahora veamos qué hace Angular una vez que se detecta un cambio:

* Angular ejecuta un algoritmo detector de cambios en cada componente de arriba a abajo en el árbol de componentes. Este algoritmo detector de cambios se genera automáticamente en tiempo de ejecución, lo que verificará y actualizará los cambios en los lugares apropiados del árbol de componentes.
* Angular es muy rápido pese a que pasa por todos los componentes de arriba a abajo para cada evento, ya que genera código amigable para VM. Debido a esto, Angular puede realizar cientos de miles de cheques en pocos milisegundos.



# Use Directives

Como entendimos el concepto de plantillas, ahora entendamos las directivas en Angular.

* Las directivas se utilizan para cambiar el comportamiento de componentes o elementos. Podemos usar directivas en forma de atributos HTML.
* Creamos directivas usando clases adjuntas con el decorator @Directive que agrega metadatos a la clase.

¿Por qué directivas?

* Las directivas modifican los elementos DOM
* Las directivas crean código reutilizable e independiente.
* Las directivas también se pueden usar para crear elementos personalizados para implementar la funcionalidad requerida

## Tipos de directivas

Hay tres tipos de directivas disponibles en Angular

* Componentes
* Directivas estructurales
* Directivas de atributos

## Componentes

* Los componentes son directivas con una plantilla o vista.
* El decorador @Component es en realidad @Directive con plantillas

## Directivas estructurales

* Una directiva estructural cambia el diseño DOM al agregar y eliminar elementos DOM.

\*directive-name = expression

* Angular tiene pocas directivas estructurales incorporadas, tales como:
  + ngIf
  + ngFor
  + ngSwitch

### ngIf

La directiva **ngIf** representa componentes o elementos condicionalmente en función de si una expresión es verdadera o falsa.

\*ngIf = "expression"

*Ejemplo en la aplicación mCart (login component).*

### ngFor

La directiva ngFor se usa para iterar sobre una colección de datos, por ejemplo: Arrays.

\*ngFor = "expression"

...

export class AppComponent {

courses: any[] = [

  { id: 1, name: "TypeScript" },

  { id: 2, name: "Angular" },

  { id: 3, name: "Node JS" },

  { id: 1, name: "TypeScript" }

];

}

<ul>

  <li \*ngFor="let course of courses; let i = index">

    {{i}} - {{ course.name }}

  </li>

</ul>

**ngFor** itera sobre el Array ***courses*** y muestra el valor de la propiedad nombre de cada curso. También almacena el índice de cada elemento en una variable llamada **i.**

***{{i}}*** muestra el índice de cada curso y ***course.name*** muestra el valor de propiedad nombre de cada curso.

### ngSwitch

* ***ngSwitch*** agrega o elimina árboles DOM cuando sus expresiones coinciden con la expresión del interruptor. Su sintaxis se compone de dos directivas, una directiva de atributos y una directiva estructural.
* Es muy similar a una declaración de cambio en JavaScript y otros lenguajes de programación.

...

export class AppComponent {

  value = 0;

  nextChoice() {

    this.value++;

  }

}

...

<div [ngSwitch]="value">

  <p \*ngSwitchCase="1">First Choice</p>

  <p \*ngSwitchCase="2">Second Choice</p>

  <p \*ngSwitchCase="3">Third Choice</p>

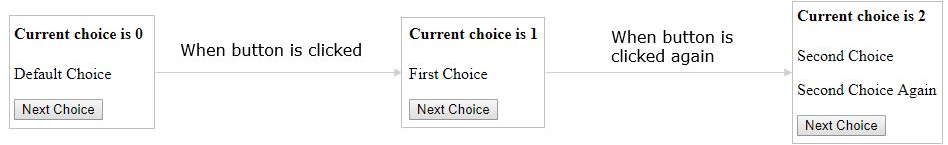
  <p \*ngSwitchCase="2">Second Choice Again</p>

  <p \*ngSwitchDefault>Default Choice</p>

</div>

<button (click)='nextChoice()'>Next Choice</button>

***ngSwitch*** toma el valor y, basándose en el valor dentro de la propiedad value, ejecuta \*ngSwitchCase. Los elementos de párrafo se agregarán / eliminarán del DOM en función del valor pasado para cambiar el case.



## Directivas Estructurales Personalizadas

Podemos crear directivas estructurales personalizadas cuando no hay una directiva integrada disponible para la funcionalidad requerida. Por ejemplo, podemos usar una directiva estructural personalizada para crear validadores personalizados para elementos de formulario.

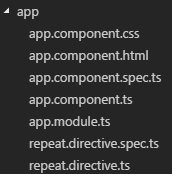
### Ejemplo:

**Planteamiento del problema:** Cree una directiva estructural personalizada llamada 'repeat' que debería repetir el elemento dado varias veces. Como no hay una directiva incorporada disponible para implementar esto, creémosla usando una directiva personalizada.

Genere una directiva llamada 'repetir' usando el siguiente comando:

***ng generate directive repeat***

Esto creará dos archivos en la carpeta src\app con los nombres repeat.directive.ts y repeat.directive.spec.ts (esto es para pruebas). Ahora la estructura de carpetas de la aplicación se verá como se muestra a continuación:



También agrega la directiva repeat al módulo raíz, es decir, **app.module.ts** para que esté disponible para todo el módulo como se muestra a continuación:

...

import { RepeatDirective } from './repeat.directive';

@NgModule({

  declarations: [

    AppComponent,

    RepeatDirective

  ],

 ...

})

export class AppModule { }

El archivo repeat.directive.ts quedará de la siguiente forma luego de agregar el código correspondiente.

1. import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
2. @Directive({
3. selector: '[appRepeat]'
4. })
5. export class RepeatDirective {
6. constructor(private \_templateRef: TemplateRef<any>,
7. private \_viewContainer: ViewContainerRef) { }
8. @Input() set appRepeat(count: number) {
9. for (var i = 0; i < count; i++) {
10. this.\_viewContainer.createEmbeddedView(this.\_templateRef);
11. }
12. }
13. }

<h3>Structural Directive</h3>

<p \*appRepeat="5">I am being repeated...</p>

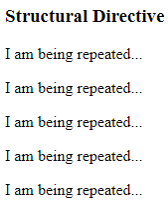
Línea 3: Anote la clase con @Directive que representa la clase como directiva y especifique el nombre del selector entre corchetes.

Línea 8: Cree un constructor e inyecte dos clases llamadas **TemplateRef** que adquieren contenido **<ng-template>** y otra clase llamada **ViewcontainerRef** que accede al contenedor HTML para agregar o eliminar elementos de él.

Línea 10: Cree un método **set** para una directiva **appRepeat** adjuntando el decorator **@Input ()** que especifica que esta directiva recibirá valor del componente. Este método toma el número pasado a la directiva **appRepeat** como argumento.

Línea 12: como necesitamos representar los elementos en función del número pasado a la directiva **appRepeat**, ejecute un bucle **for** en el que pase la referencia de plantilla a un método **createEmbeddedView** que muestre los elementos en el **DOM**. Esta directiva estructural crea una vista incrustada a partir de la **<ng-template>** generada por Angular e inserta esa vista en un contenedor de vista.

En el ejemplo se está pasando un 5 a la directiva, por lo que el resultado pintará 5 veces el texto.



### Propiedad exportAs

La propiedad **exportAs** se utiliza para definir un nombre para una directiva mediante la cual podemos acceder a las propiedades y métodos de clase de una directiva en una plantilla de componente.

La propiedad **exportAs** se puede aplicar a componentes y directivas.

1. import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
2. @Directive({
3. selector: '[appRepeat]',
4. exportAs: 'repeat,changeText'
5. })
6. export class RepeatDirective {
7. constructor(private \_templateRef: TemplateRef<any>, private \_viewContainer: ViewContainerRef) { }
8. repeatElement(count: number) {
9. for (let i = 0; i < count; i++) {
10. this.\_viewContainer.createEmbeddedView(this.\_templateRef);
11. }
12. }
13. changeElementText(count: number) {
14. for (let i = 0; i < 5; i++) {
15. document.getElementsByTagName('p').item(i).innerHTML = 'Text is changed...';
16. }
17. }
18. }

Línea 5: la directiva **appRepeat** se exporta con dos nombres aquí, es decir, **repeat** y **changeText**. Podemos usar varios nombres para exportar una directiva.

Línea 11: el método **repeatElement()** tomará un valor de conteo como entrada y creará y representará los elementos en la página.

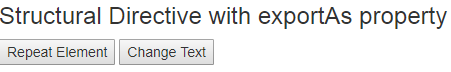
Línea 17: el método **changeElementText()** tomará un valor de conteo como entrada y cambiará el texto de los elementos HTML representados.

1. <h3>Structural Directive with exportAs property</h3>
2. <ng-template appRepeat #rd="repeat" #ct="changeText">
3. <p>I am being repeated...</p>
4. </ng-template>
5. <button (click)="rd.repeatElement(5)">Repeat Element</button>
6. <button (click)="ct.changeElementText(5)">Change Text</button>

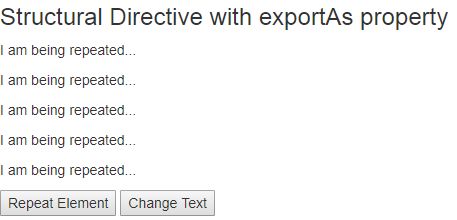
Línea 3: Aquí la sección de la plantilla está ligada (binded) con las variables de plantilla **'rd'** y **'ct'**. Cada variable de plantilla se ha asignado con los nombres exportados de una clase directiva. Ahora en una plantilla, podemos acceder a los métodos de clase de directiva usando estas variables de plantilla llamadas **'rd'** y **'ct'** donde cada una es una referencia a los nombres **'repeat'** y **'changeText'** de una directiva.

Línea 7: se accede al método repeatElement () de una clase directiva utilizando la variable de template **rd**. Cuando se hace clic en este botón, invoca un método que, a su vez, muestra cinco párrafos en la página.

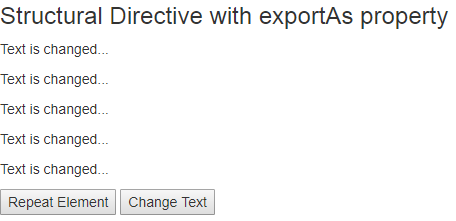
Línea 8: se accede al método changeElementText () de una clase directiva utilizando la variable de template **ct**. Cuando se hace clic en este botón, invoca el método que, a su vez, cambiará el texto de todos los párrafos representados.



Cuando Repeat Element es cliqueado:



Cuando Change Text es cliqueado:



## Directivas de Atributos

Las directivas de atributos cambian la apariencia / comportamiento de un componente / elemento

Las siguientes son directivas de atributo incorporadas

* ngStyle
* ngClass

### ngStyle

Esta directiva se utiliza para modificar el estilo de un componente / elemento. Podemos usar la siguiente sintaxis para establecer un estilo CSS único para el elemento que también se conoce como style binding (enlace de estilo).

[style.<cssproperty>] = "value"

Ejemplo:

export class AppComponent {

  colorName = 'yellow';

  color = 'red';

}

<div [style.background-color]="colorName" [style.color]="color">

  Uses fixed yellow background

</div>

**style.background-color** establecerá el color de fondo del texto en amarillo y la directiva **style.color** establecerá el color del texto en rojo.

Si hay más de un estilo CSS para aplicar, podemos usar el atributo ngStyle.

...

export class AppComponent {

  colorName = 'red';

  fontWeight = 'bold';

  borderStyle = '1px solid black';

}

<p [ngStyle]="{

                color:colorName,

                'font-weight':fontWeight,

                borderBottom: borderStyle

            }">

  Demo for attribute directive ngStyle

</p>

La directiva **NgStyle** se usa aquí para establecer múltiples estilos CSS para el texto dado. ***Como nota adicional, cuando el style property tiene un guión se debe utilizar comilla simple.***

### ngClass

Nos permite establecer y cambiar **dinámicamente** las clases CSS para un elemento DOM dado. Podemos usar la siguiente sintaxis para establecer una sola clase css para el elemento que también se conoce como class binding (enlace de clase).

[class.css\_class\_name] = "property/value"

Ts file:

export class AppComponent {

  isBordered = true;

}

HTML file:

<div [class.bordered]="isBordered">

  Border {{ isBordered ? "ON" : "OFF" }}

</div>

CSS file:

.bordered {

    border: 1px dashed black;

    background-color: #eee;

}

**Bind** (enlaza) la propiedad **isBordered** con la clase CSS **bordered**. La clase CSS **bordered** solo se aplicará si la propiedad **isBordered** se evalúa como verdadera.

Si tenemos más de una clase CSS para aplicar, entonces buscaremos la sintaxis **ngClass.**

[ngClass] = "{css\_class\_name1 : Boolean expression, css\_class\_name2: Boolean expression, ……}"

Ejemplo:

TS File:

export class AppComponent  {

   isBordered = true;

   isColor = true;

}

HTML File:

<div [ngClass]="{bordered: isBordered, color: isColor}">

  Border {{ isBordered ? "ON" : "OFF" }}

</div>

CSS File:

.bordered {

    border: 1px dashed black;

    background-color: #eee;

}

.color {

    color: blue;

}

Dos clases CSS llamadas **bordered** y **color** se aplican a un tag **div**. Ambas clases están vinculadas con las propiedades **isBordered** e **isColor**, donde las clases CSS se aplicarán al tag **div** solo si las propiedades devuelven **true.**

# Implement data binding

# Apply pipes

# Create nested components and share data between them

# Add validations to the forms

# Create services

# Implement routing