Guía de Angular

1. ¿Qué es Angular? - Herramientas necesarias……………………………………………………………………………………………. 4
2. Creación de un proyecto y prueba de su funcionamiento…………………………………………………………………………. 7
3. Archivos y carpetas básicas de un proyecto en Angular……………………………………………………………………….……. 10
4. Interpolación en los archivos HTML de Angular……………………………………………………………………………………..…. 24
5. Sintaxis de Template para estructuras condicionales y repetitivas: @if / @else - @for - @switch/@case/@default... 28
6. Captura de eventos………………………………………………………………………………………………………………………………..…. 33
7. Enlace de propiedades (Property Binding) ……………………………………………………………………………………….………. 36
8. Componentes: creación……………………………………………………………………………………………………………………………. 38
9. Componentes: pasar datos del componente padre al componente hijo……………………………………………………. 45
10. Componentes: disparo de eventos de la componente hija a la componente padre…………..………………………. 50
11. Componentes: llamar a métodos del componente hijo desde el template del padre………………………………... 54
12. Componentes: llamar a métodos del componente hijo desde la clase padre……………………………………………. 58
13. Formularios: formulario basado en plantillas……………………………………………………………………………………………. 62
14. Formularios reactivos : ReactiveFormsModule, FormControl……………………………………………………………………. 73
15. Formularios reactivos: FormGroup………………………………………………………………………………………………………..…. 79
16. Formularios reactivos : controles checkbox, radio y select……………………………..…………………………..……………. 82
17. Formularios reactivos : FormGroup anidados………………………………………………………………………………………..…. 86
18. Formularios reactivos : validaciones estándares de Angular……………………………………………………..………………. 10
19. Formularios reactivos : validaciones personalizadas………………………………………………………………………………..…10
20. Formularios reactivos : FormBuilder…………………………………………………………………………………………………………. 10
21. Router : definición de rutas………………………………………………………………………………………………………………………. 10
22. Router : parámetros en una ruta………………………………………………………………………………………………………………. 10
23. Router : captura de ruta no existente………………………………………………………………………………………….……………. 10
24. Router : rutas anidadas……………………………………………………………………………………………………………………….……. 10
25. Servicios : concepto y pasos para crearlos………………………………………………………………..………………………………. 10
26. Servicios : recuperación de datos de un servidor web………………………………………………………………………………. 10
27. Comunicación entre Angular y PHP……………………………………………………………………………………..……………………. 10
28. Pipes: definición…………………………………………………………………………………………………………………………….…………. 10
29. Pipes: creación de pipes personalizadas………………………………………………………………………………………..…………. 110
30. Compilación y despliegue de una aplicación Angular en un servidor de Internet………………………………………. 10
31. Herramienta Angular CLI………………………………………………………………………..…………………………………………………. 10
32. Herramienta Angular CLI - comando: ng new……………………………………………………………………………………………. 10
33. Herramienta Angular CLI - comando: ng generate……………………………………………….……………………………………. 10
34. Herramienta Angular CLI - comando: ng serve………………………………………………………….………………………………. 10
35. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta raiz…………………………………. 10
36. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta 'src'……………………………..…. 10
37. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta 'app'………………………………. 10
38. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta 'public'……………………..……. 10
39. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpetas '<Nombre del proyecto>\dist'.……… 10
40. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta node\_modules………….……. 10
41. Vistas diferidas : @defer {} @placeholder{}………………………………………………………………………………………………. 10
42. Vistas diferidas : @defer ([opciones de activación]) {}………………………………………………………………………………. 10
43. Vistas diferidas : Lazy Loading (carga perezosa) …………………………………………………………………………………….…. 10
44. Vistas diferidas : @defer (when [expresión lógica]) {}……………………………………………………………………………….. 10
45. Vistas diferidas : @defer {} @placeholder{} @loading{} @error{}………………………………………………………..……. 10
46. Directiva de atributo [ngStyle] y directivas para estilos individuales - Sintaxis completa………………..…………. 10
47. Directiva de atributo [ngClass] y directivas para clases individuales - Sintaxis completa……………………………. 10
48. Directivas de atributo - creación de directivas personalizadas……………………………………………….…………………. 10
49. Directivas de atributo - creación y definición de propiedades………..…………………………………………………………. 10
50. Directivas de atributo - responder a eventos del usuario dentro de la directiva…………………………….…………. 10
51. TypeScript………………………………………………………………………………………………………………………………..………………. 10
52. TypeScript: tipado estático………………………………………………………………………………………….……………………………. 10
53. TypeScript: clases………………………………………………………………………………………………………………..……………………. 10
54. TypeScript: funciones y métodos………………………………………………………………………………………………………………. 10
55. TypeScript: herencia…………………………………………………………………………………………………………………………………. 10
56. TypeScript: interfaces…………………………………………………………………………………………………….…………………………. 10
57. TypeScript: clases genéricas………………………………………………………………………………………..……………………………. 10
58. TypeScript y Angular: definición de una interfaz para controlar los tipos………………………………………….………. 10
59. TypeScript: decoradores……………………………………………………………………………………..……………………………………. 10
60. TypeScript y Angular: decorador @Component………………………………………………………………………..………………. 10
61. TypeScript y Angular: decoradores @Input y @Output…………………………………….………………………………………. 10
62. TypeScript y Angular: decorador @Injectable…………………………………………………………………..………………………. 10
63. TypeScript y Angular: decorador @Directive…………………………………………………………………………………….………. 10
64. TypeScript y Angular: decorador @Pipe……………………………………………………………………………………………………. 10
65. Angular e instalación de Bootstrap original………………………………………………………………………………………………. 10
66. Angular y Bootstrap - creación de una componente: barrademenu……………………….…………………………………. 10
67. Angular y Bootstrap - creación de una componente: alerta………………………………………………………………………. 10
68. Angular y Bootstrap - creación de una componente: tabla………………………….……………………………………………. 10
69. Angular y Bootstrap - creación de una componente: tarjeta…………………….………………………………………………. 10
70. Consumir una API pública utilizando una interface para definir su estructura y la clase HttpClient……..……. 10
71. Que utilizar en Angular: Promise u Observable…………………………………………………………………………………………. 10
72. ¿Qué es Angular? - Herramientas necesarias

Angular es un framework para el desarrollo de aplicaciones web. Está pensado para dividir un proyecto en componentes y ser reutilizadas en proyectos medianos y grandes.

Sus principales competidores son [Vue](https://vuejs.org/) (proyecto iniciado por Evan You en 2014) y [React](https://reactjs.org/) (proyecto iniciado por Facebook en 2013)

La última versión (19) introducida a fines de 2024 presenta cambios muy importantes con las 16 versiones anteriores, a tal punto que ha creado un nuevo sitio web para empezar desde cero (sitio presentado a fines de 2023 con la versión 17): [Angular Dev](https://angular.dev/)

La versión anterior de [Angular](https://angular.io/) tiene su salida al mercado en 2016 pero además tiene una versión previa no compatible y solo mantenida para proyectos antiguos llamada [Angular.js](https://angularjs.org/) (2010)

Si tiene que trabajar con proyectos legacy puede tal vez tener la necesidad de ver el [tutorial de Angular previo a la versión 17](https://www.tutorialesprogramacionya.com/angularya/)

El proyecto de Angular es propiedad de la empresa de Google.

La versión de Angular que trabajamos en este tutorial es la 19 (salió el 19/11/2024)

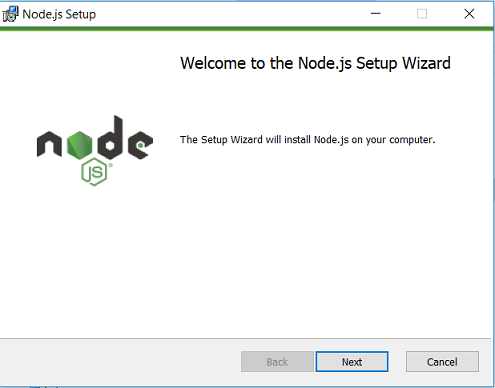
Para desarrollar en forma efectiva una aplicación en Angular debemos instalar al menos dos herramientas básicas:

* Node.js
* Angular CLI (Command Line Interface - Interfaz de línea de comandos)

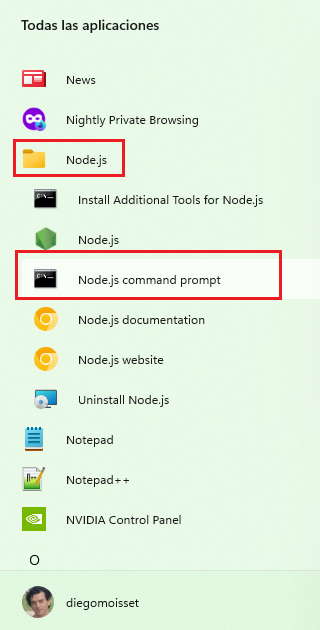
Instalación de Node.js

La primera herramienta a instalar será Node.js, esto debido a que gran cantidad de programas para el desarrollo en Angular están implementadas en Node.

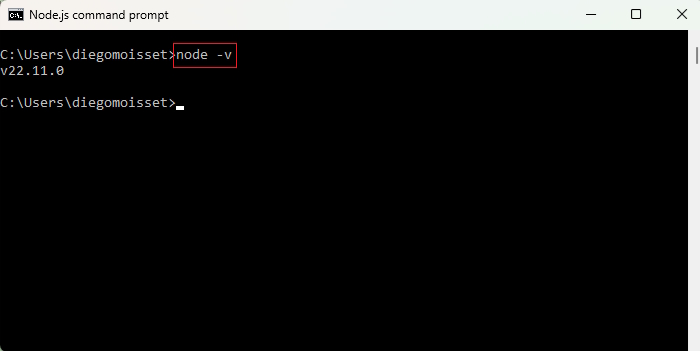
Debemos Descargar e instalar la última versión estable de [Node.js](https://nodejs.org/en/):



Una vez instalado debemos ingresar a la línea de comandos que nos provee Nodo.js:



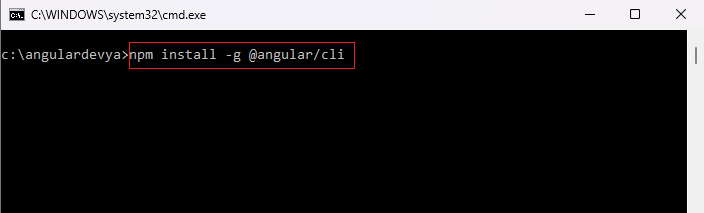
Para comprobar su correcto funcionamiento podemos averiguar su versión:



Instalación de Angular CLI

Para instalar este software lo hacemos desde la misma línea de comandos de Node.js (por eso lo instalamos primero), debemos ejecutar el siguiente comando:

npm install -g @angular/cli



Es importante el -g para que se instale en forma global.

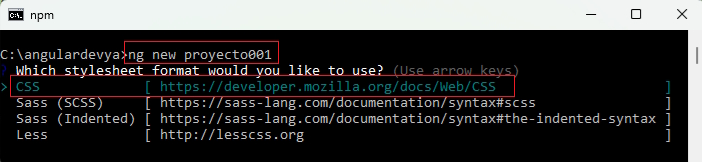
1. Creación de un proyecto y prueba de su funcionamiento

Para crear un proyecto vamos a utilizar la aplicación Angular CLI que acabamos de instalar en el concepto anterior.

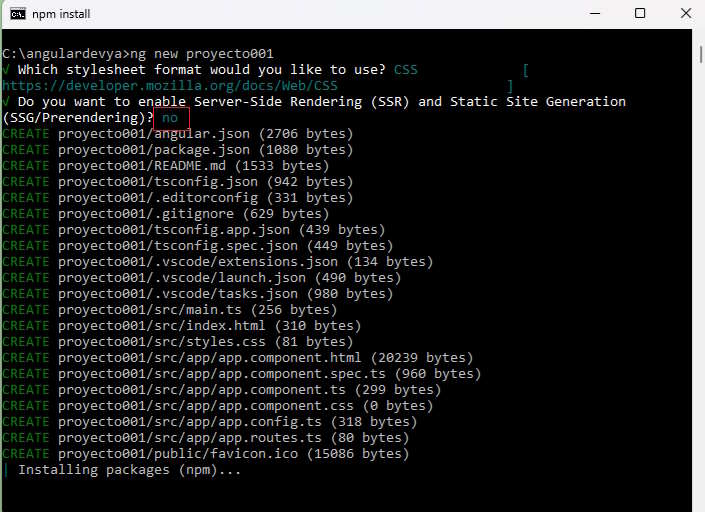
Desde la línea de comandos de Node.js procedemos a ejecutar el siguiente comando:

ng new proyecto001

Seleccionaremos que utilizaremos archivos CSS para los estilos (valor seleccionado por defecto), presionamos la tecla "entrada":



Seguidamente nos consulta si queremos generar "Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)", por el momento dejamos por defecto que No (es una característica agregada a Angular a partir de la versión 17):

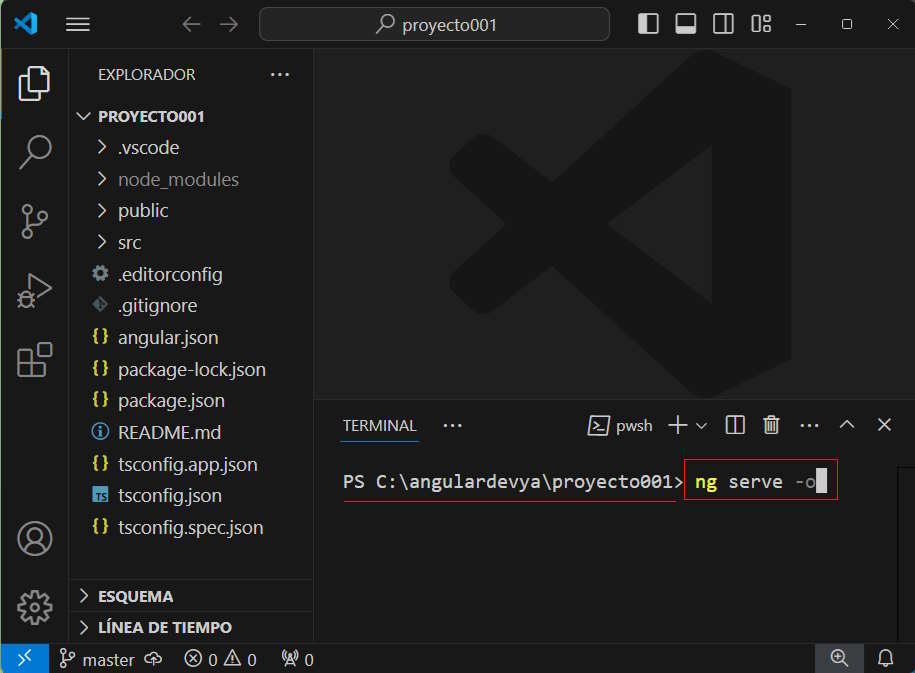


El comando "ng new" crea la carpeta proyecto001 e instala una gran cantidad de herramientas que nos auxiliarán durante el desarrollo del proyecto (255 MB). Como Angular esta pensado para aplicaciones de complejidad media o alta no hay posibilidad de instalar menos herramientas.

El proceso de generar el proyecto lleva bastante tiempo ya que deben descargarse de internet muchas herramientas.

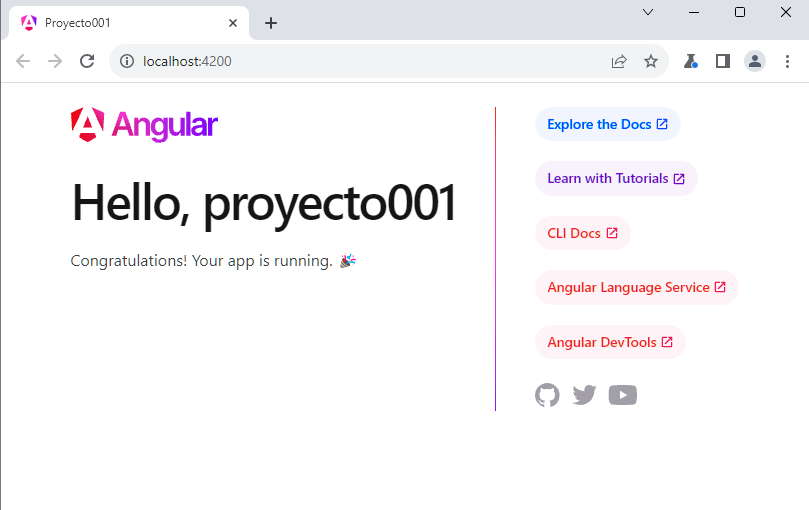
Se genera una aplicación con el esqueleto mínimo, para probarlo podemos abrir el Visual Studio Code y debemos ir a la carpeta que se acaba de crear y lanzar el siguiente comando:

ng serve -o



Este comando arranca un servidor web en forma local y abre el navegador para la ejecución de la aplicación.

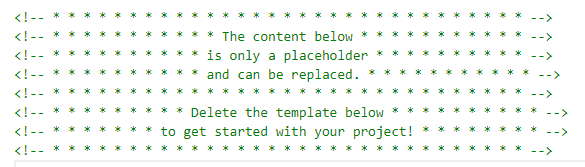
En el navegador tenemos como resultado:



En este concepto no me interesa ver todos las carpetas y archivos generados. Solo efectuaremos un cambio para ver como se reflejan en el navegador.

En la carpeta 'proyecto001' hay una subcarpeta llamada 'src' y dentro de esta una llamada 'app', busquemos el archivo 'app.component.html' y procedamos a borrar todo menos la etiqueta <router-outlet />.

Si leemos las primeras líneas nos informa que siempre debemos modificar el archivo con los algoritmos de nuestro proyecto (se genera a modo de ejemplo):



Disponemos el siguiente código remplazando al generado en forma automática:

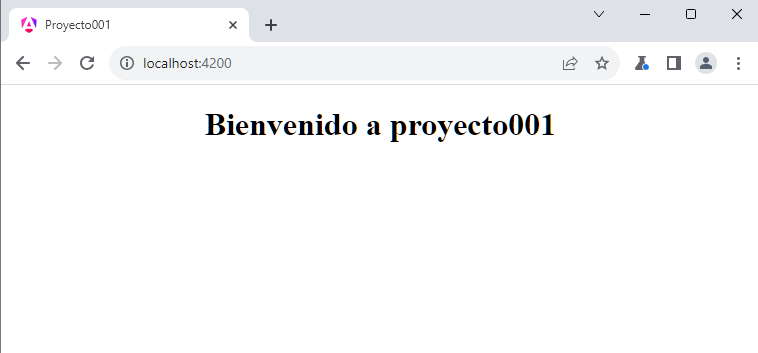
<h1 style="text-align:center">

Bienvenido a {{ title }}

</h1>

<router-outlet />

Una vez que grabamos los cambios en este archivo podemos ver que automáticamente se ven reflejados en el navegador (recordemos de no cerrar la consola del Visual Studio Code donde ejecutamos el comando "ng serve -o"):



A partir del próximo concepto comenzaremos a analizar un proyecto Angular, en este momento me interesa solo recordar los pasos que debemos dar para crear y ejecutar un proyecto.

1. Archivos y carpetas básicas de un proyecto en Angular

Vimos en el concepto anterior que para crear un proyecto en Angular utilizamos la herramienta Angular CLI y desde la línea de comandos escribimos:

ng new proyecto001

No haremos por el momento un estudio exhaustivo de todos los archivos y carpetas que se crean (más 25.900 archivos y 3300 carpetas en la versión de Angular 19.x), sino de aquellas que se requieren modificar según el concepto que estemos estudiando.

En Angular la pieza fundamental es la 'COMPONENTE'. Debemos pensar siempre que una aplicación se construye a base de un conjunto de componentes (por ejemplo pueden ser componentes: un menú, lista de usuarios, login, tabla de datos, calendario, formulario de búsqueda etc.)

Angular CLI nos crea una única componente llamada 'AppComponent' que se distribuye en 4 archivos:

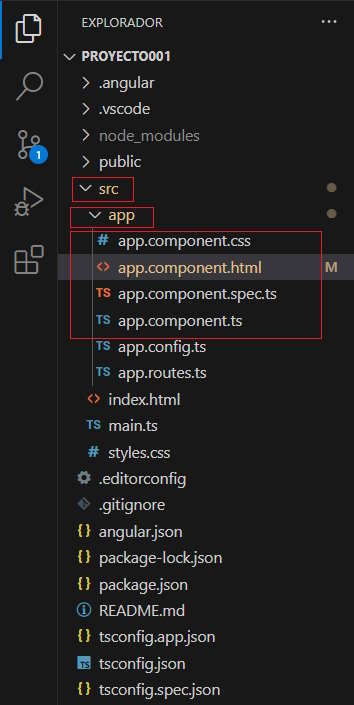
app.component.ts

app.component.html

app.component.css

app.component.spec.ts

Todos estos archivos se localizan en la carpeta 'app' y esta carpeta se encuentra dentro de la carpeta 'src':



En Angular se programa utilizando el lenguaje TypeScript que vamos a ir aprendiéndolo a lo largo del curso. El archivo donde se declara la clase AppComponent es 'app.component.ts':

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

title = 'proyecto001';

}

La Clase AppComponent define un atributo llamado 'title' y lo inicializa con el string 'proyecto001' que coincide con el nombre del proyecto que creamos:

title = 'proyecto001';

Dijimos anteriormente que la clase completa se distribuye en otros archivos y podemos ver que mediante la función decoradora @Component le indicamos los otros archivos que pertenecen a esta componente:

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

El archivo 'app.component.html' tiene la parte visual de nuestra componente 'AppComponent' y está constituido mayormente por código HTML (cada vez que realicemos un proyecto a este código lo borraremos para resolver nuestro problema, salvo la etiqueta <router-outlet />):

<!-- \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* \* The content below \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* is only a placeholder \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* and can be replaced. \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* Delete the template below \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* to get started with your project! \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* -->

<style>

:host {

--bright-blue: oklch(51.01% 0.274 263.83);

--electric-violet: oklch(53.18% 0.28 296.97);

--french-violet: oklch(47.66% 0.246 305.88);

--vivid-pink: oklch(69.02% 0.277 332.77);

--hot-red: oklch(61.42% 0.238 15.34);

--orange-red: oklch(63.32% 0.24 31.68);

--gray-900: oklch(19.37% 0.006 300.98);

--gray-700: oklch(36.98% 0.014 302.71);

--gray-400: oklch(70.9% 0.015 304.04);

--red-to-pink-to-purple-vertical-gradient: linear-gradient(

180deg,

var(--orange-red) 0%,

var(--vivid-pink) 50%,

var(--electric-violet) 100%

);

--red-to-pink-to-purple-horizontal-gradient: linear-gradient(

90deg,

var(--orange-red) 0%,

var(--vivid-pink) 50%,

var(--electric-violet) 100%

);

--pill-accent: var(--bright-blue);

font-family: "Inter", -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto,

Helvetica, Arial, sans-serif, "Apple Color Emoji", "Segoe UI Emoji",

"Segoe UI Symbol";

box-sizing: border-box;

-webkit-font-smoothing: antialiased;

-moz-osx-font-smoothing: grayscale;

}

h1 {

font-size: 3.125rem;

color: var(--gray-900);

font-weight: 500;

line-height: 100%;

letter-spacing: -0.125rem;

margin: 0;

font-family: "Inter Tight", -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto,

Helvetica, Arial, sans-serif, "Apple Color Emoji", "Segoe UI Emoji",

"Segoe UI Symbol";

}

p {

margin: 0;

color: var(--gray-700);

}

main {

width: 100%;

min-height: 100%;

display: flex;

justify-content: center;

align-items: center;

padding: 1rem;

box-sizing: inherit;

position: relative;

}

.angular-logo {

max-width: 9.2rem;

}

.content {

display: flex;

justify-content: space-around;

width: 100%;

max-width: 700px;

margin-bottom: 3rem;

}

.content h1 {

margin-top: 1.75rem;

}

.content p {

margin-top: 1.5rem;

}

.divider {

width: 1px;

background: var(--red-to-pink-to-purple-vertical-gradient);

margin-inline: 0.5rem;

}

.pill-group {

display: flex;

flex-direction: column;

align-items: start;

flex-wrap: wrap;

gap: 1.25rem;

}

.pill {

display: flex;

align-items: center;

--pill-accent: var(--bright-blue);

background: color-mix(in srgb, var(--pill-accent) 5%, transparent);

color: var(--pill-accent);

padding-inline: 0.75rem;

padding-block: 0.375rem;

border-radius: 2.75rem;

border: 0;

transition: background 0.3s ease;

font-family: var(--inter-font);

font-size: 0.875rem;

font-style: normal;

font-weight: 500;

line-height: 1.4rem;

letter-spacing: -0.00875rem;

text-decoration: none;

}

.pill:hover {

background: color-mix(in srgb, var(--pill-accent) 15%, transparent);

}

.pill-group .pill:nth-child(6n + 1) {

--pill-accent: var(--bright-blue);

}

.pill-group .pill:nth-child(6n + 2) {

--pill-accent: var(--french-violet);

}

.pill-group .pill:nth-child(6n + 3),

.pill-group .pill:nth-child(6n + 4),

.pill-group .pill:nth-child(6n + 5) {

--pill-accent: var(--hot-red);

}

.pill-group svg {

margin-inline-start: 0.25rem;

}

.social-links {

display: flex;

align-items: center;

gap: 0.73rem;

margin-top: 1.5rem;

}

.social-links path {

transition: fill 0.3s ease;

fill: var(--gray-400);

}

.social-links a:hover svg path {

fill: var(--gray-900);

}

@media screen and (max-width: 650px) {

.content {

flex-direction: column;

width: max-content;

}

.divider {

height: 1px;

width: 100%;

background: var(--red-to-pink-to-purple-horizontal-gradient);

margin-block: 1.5rem;

}

}

</style>

<main class="main">

<div class="content">

<div class="left-side">

<svg

xmlns="http://www.w3.org/2000/svg"

viewBox="0 0 982 239"

fill="none"

class="angular-logo"

>

<g clip-path="url(#a)">

<path

fill="url(#b)"

d="M388.676 191.625h30.849L363.31 31.828h-35.758l-56.215 159.797h30.848l13.174-39.356h60.061l13.256 39.356Zm-65.461-62.675 21.602-64.311h1.227l21.602 64.311h-44.431Zm126.831-7.527v70.202h-28.23V71.839h27.002v20.374h1.392c2.782-6.71 7.2-12.028 13.255-15.956 6.056-3.927 13.584-5.89 22.503-5.89 8.264 0 15.465 1.8 21.684 5.318 6.137 3.518 10.964 8.673 14.319 15.382 3.437 6.71 5.074 14.81 4.992 24.383v76.175h-28.23v-71.92c0-8.019-2.046-14.237-6.219-18.819-4.173-4.5-9.819-6.791-17.102-6.791-4.91 0-9.328 1.063-13.174 3.272-3.846 2.128-6.792 5.237-9.001 9.328-2.046 4.009-3.191 8.918-3.191 14.728ZM589.233 239c-10.147 0-18.82-1.391-26.103-4.091-7.282-2.7-13.092-6.382-17.511-10.964-4.418-4.582-7.528-9.655-9.164-15.219l25.448-6.136c1.145 2.372 2.782 4.663 4.991 6.954 2.209 2.291 5.155 4.255 8.837 5.81 3.683 1.554 8.428 2.291 14.074 2.291 8.019 0 14.647-1.964 19.884-5.81 5.237-3.845 7.856-10.227 7.856-19.064v-22.665h-1.391c-1.473 2.946-3.601 5.892-6.383 9.001-2.782 3.109-6.464 5.645-10.965 7.691-4.582 2.046-10.228 3.109-17.101 3.109-9.165 0-17.511-2.209-25.039-6.545-7.446-4.337-13.42-10.883-17.757-19.474-4.418-8.673-6.628-19.473-6.628-32.565 0-13.091 2.21-24.301 6.628-33.383 4.419-9.082 10.311-15.955 17.839-20.7 7.528-4.746 15.874-7.037 25.039-7.037 7.037 0 12.846 1.145 17.347 3.518 4.582 2.373 8.182 5.236 10.883 8.51 2.7 3.272 4.746 6.382 6.137 9.327h1.554v-19.8h27.821v121.749c0 10.228-2.454 18.737-7.364 25.447-4.91 6.709-11.538 11.7-20.048 15.055-8.509 3.355-18.165 4.991-28.884 4.991Zm.245-71.266c5.974 0 11.047-1.473 15.302-4.337 4.173-2.945 7.446-7.118 9.573-12.519 2.21-5.482 3.274-12.027 3.274-19.637 0-7.609-1.064-14.155-3.274-19.8-2.127-5.646-5.318-10.064-9.491-13.255-4.174-3.11-9.329-4.746-15.384-4.746s-11.537 1.636-15.792 4.91c-4.173 3.272-7.365 7.772-9.492 13.418-2.128 5.727-3.191 12.191-3.191 19.392 0 7.2 1.063 13.745 3.273 19.228 2.127 5.482 5.318 9.736 9.573 12.764 4.174 3.027 9.41 4.582 15.629 4.582Zm141.56-26.51V71.839h28.23v119.786h-27.412v-21.273h-1.227c-2.7 6.709-7.119 12.191-13.338 16.446-6.137 4.255-13.747 6.382-22.748 6.382-7.855 0-14.81-1.718-20.783-5.237-5.974-3.518-10.72-8.591-14.075-15.382-3.355-6.709-5.073-14.891-5.073-24.464V71.839h28.312v71.921c0 7.609 2.046 13.664 6.219 18.083 4.173 4.5 9.655 6.709 16.365 6.709 4.173 0 8.183-.982 12.111-3.028 3.927-2.045 7.118-5.072 9.655-9.082 2.537-4.091 3.764-9.164 3.764-15.218Zm65.707-109.395v159.796h-28.23V31.828h28.23Zm44.841 162.169c-7.61 0-14.402-1.391-20.457-4.091-6.055-2.7-10.883-6.791-14.32-12.109-3.518-5.319-5.237-11.946-5.237-19.801 0-6.791 1.228-12.355 3.765-16.773 2.536-4.419 5.891-7.937 10.228-10.637 4.337-2.618 9.164-4.664 14.647-6.055 5.4-1.391 11.046-2.373 16.856-3.027 7.037-.737 12.683-1.391 17.102-1.964 4.337-.573 7.528-1.555 9.574-2.782 1.963-1.309 3.027-3.273 3.027-5.973v-.491c0-5.891-1.718-10.391-5.237-13.664-3.518-3.191-8.51-4.828-15.056-4.828-6.955 0-12.356 1.473-16.447 4.5-4.009 3.028-6.71 6.546-8.183 10.719l-26.348-3.764c2.046-7.282 5.483-13.336 10.31-18.328 4.746-4.909 10.638-8.59 17.511-11.045 6.955-2.455 14.565-3.682 22.912-3.682 5.809 0 11.537.654 17.265 2.045s10.965 3.6 15.711 6.71c4.746 3.109 8.51 7.282 11.455 12.6 2.864 5.318 4.337 11.946 4.337 19.883v80.184h-27.166v-16.446h-.9c-1.719 3.355-4.092 6.464-7.201 9.328-3.109 2.864-6.955 5.237-11.619 6.955-4.828 1.718-10.229 2.536-16.529 2.536Zm7.364-20.701c5.646 0 10.556-1.145 14.729-3.354 4.173-2.291 7.364-5.237 9.655-9.001 2.292-3.763 3.355-7.854 3.355-12.273v-14.155c-.9.737-2.373 1.391-4.5 2.046-2.128.654-4.419 1.145-7.037 1.636-2.619.491-5.155.9-7.692 1.227-2.537.328-4.746.655-6.628.901-4.173.572-8.019 1.472-11.292 2.781-3.355 1.31-5.973 3.11-7.855 5.401-1.964 2.291-2.864 5.318-2.864 8.918 0 5.237 1.882 9.164 5.728 11.782 3.682 2.782 8.51 4.091 14.401 4.091Zm64.643 18.328V71.839h27.412v19.965h1.227c2.21-6.955 5.974-12.274 11.292-16.038 5.319-3.763 11.456-5.645 18.329-5.645 1.555 0 3.355.082 5.237.163 1.964.164 3.601.328 4.91.573v25.938c-1.227-.41-3.109-.819-5.646-1.146a58.814 58.814 0 0 0-7.446-.49c-5.155 0-9.738 1.145-13.829 3.354-4.091 2.209-7.282 5.236-9.655 9.164-2.373 3.927-3.519 8.427-3.519 13.5v70.448h-28.312ZM222.077 39.192l-8.019 125.923L137.387 0l84.69 39.192Zm-53.105 162.825-57.933 33.056-57.934-33.056 11.783-28.556h92.301l11.783 28.556ZM111.039 62.675l30.357 73.803H80.681l30.358-73.803ZM7.937 165.115 0 39.192 84.69 0 7.937 165.115Z"

/>

<path

fill="url(#c)"

d="M388.676 191.625h30.849L363.31 31.828h-35.758l-56.215 159.797h30.848l13.174-39.356h60.061l13.256 39.356Zm-65.461-62.675 21.602-64.311h1.227l21.602 64.311h-44.431Zm126.831-7.527v70.202h-28.23V71.839h27.002v20.374h1.392c2.782-6.71 7.2-12.028 13.255-15.956 6.056-3.927 13.584-5.89 22.503-5.89 8.264 0 15.465 1.8 21.684 5.318 6.137 3.518 10.964 8.673 14.319 15.382 3.437 6.71 5.074 14.81 4.992 24.383v76.175h-28.23v-71.92c0-8.019-2.046-14.237-6.219-18.819-4.173-4.5-9.819-6.791-17.102-6.791-4.91 0-9.328 1.063-13.174 3.272-3.846 2.128-6.792 5.237-9.001 9.328-2.046 4.009-3.191 8.918-3.191 14.728ZM589.233 239c-10.147 0-18.82-1.391-26.103-4.091-7.282-2.7-13.092-6.382-17.511-10.964-4.418-4.582-7.528-9.655-9.164-15.219l25.448-6.136c1.145 2.372 2.782 4.663 4.991 6.954 2.209 2.291 5.155 4.255 8.837 5.81 3.683 1.554 8.428 2.291 14.074 2.291 8.019 0 14.647-1.964 19.884-5.81 5.237-3.845 7.856-10.227 7.856-19.064v-22.665h-1.391c-1.473 2.946-3.601 5.892-6.383 9.001-2.782 3.109-6.464 5.645-10.965 7.691-4.582 2.046-10.228 3.109-17.101 3.109-9.165 0-17.511-2.209-25.039-6.545-7.446-4.337-13.42-10.883-17.757-19.474-4.418-8.673-6.628-19.473-6.628-32.565 0-13.091 2.21-24.301 6.628-33.383 4.419-9.082 10.311-15.955 17.839-20.7 7.528-4.746 15.874-7.037 25.039-7.037 7.037 0 12.846 1.145 17.347 3.518 4.582 2.373 8.182 5.236 10.883 8.51 2.7 3.272 4.746 6.382 6.137 9.327h1.554v-19.8h27.821v121.749c0 10.228-2.454 18.737-7.364 25.447-4.91 6.709-11.538 11.7-20.048 15.055-8.509 3.355-18.165 4.991-28.884 4.991Zm.245-71.266c5.974 0 11.047-1.473 15.302-4.337 4.173-2.945 7.446-7.118 9.573-12.519 2.21-5.482 3.274-12.027 3.274-19.637 0-7.609-1.064-14.155-3.274-19.8-2.127-5.646-5.318-10.064-9.491-13.255-4.174-3.11-9.329-4.746-15.384-4.746s-11.537 1.636-15.792 4.91c-4.173 3.272-7.365 7.772-9.492 13.418-2.128 5.727-3.191 12.191-3.191 19.392 0 7.2 1.063 13.745 3.273 19.228 2.127 5.482 5.318 9.736 9.573 12.764 4.174 3.027 9.41 4.582 15.629 4.582Zm141.56-26.51V71.839h28.23v119.786h-27.412v-21.273h-1.227c-2.7 6.709-7.119 12.191-13.338 16.446-6.137 4.255-13.747 6.382-22.748 6.382-7.855 0-14.81-1.718-20.783-5.237-5.974-3.518-10.72-8.591-14.075-15.382-3.355-6.709-5.073-14.891-5.073-24.464V71.839h28.312v71.921c0 7.609 2.046 13.664 6.219 18.083 4.173 4.5 9.655 6.709 16.365 6.709 4.173 0 8.183-.982 12.111-3.028 3.927-2.045 7.118-5.072 9.655-9.082 2.537-4.091 3.764-9.164 3.764-15.218Zm65.707-109.395v159.796h-28.23V31.828h28.23Zm44.841 162.169c-7.61 0-14.402-1.391-20.457-4.091-6.055-2.7-10.883-6.791-14.32-12.109-3.518-5.319-5.237-11.946-5.237-19.801 0-6.791 1.228-12.355 3.765-16.773 2.536-4.419 5.891-7.937 10.228-10.637 4.337-2.618 9.164-4.664 14.647-6.055 5.4-1.391 11.046-2.373 16.856-3.027 7.037-.737 12.683-1.391 17.102-1.964 4.337-.573 7.528-1.555 9.574-2.782 1.963-1.309 3.027-3.273 3.027-5.973v-.491c0-5.891-1.718-10.391-5.237-13.664-3.518-3.191-8.51-4.828-15.056-4.828-6.955 0-12.356 1.473-16.447 4.5-4.009 3.028-6.71 6.546-8.183 10.719l-26.348-3.764c2.046-7.282 5.483-13.336 10.31-18.328 4.746-4.909 10.638-8.59 17.511-11.045 6.955-2.455 14.565-3.682 22.912-3.682 5.809 0 11.537.654 17.265 2.045s10.965 3.6 15.711 6.71c4.746 3.109 8.51 7.282 11.455 12.6 2.864 5.318 4.337 11.946 4.337 19.883v80.184h-27.166v-16.446h-.9c-1.719 3.355-4.092 6.464-7.201 9.328-3.109 2.864-6.955 5.237-11.619 6.955-4.828 1.718-10.229 2.536-16.529 2.536Zm7.364-20.701c5.646 0 10.556-1.145 14.729-3.354 4.173-2.291 7.364-5.237 9.655-9.001 2.292-3.763 3.355-7.854 3.355-12.273v-14.155c-.9.737-2.373 1.391-4.5 2.046-2.128.654-4.419 1.145-7.037 1.636-2.619.491-5.155.9-7.692 1.227-2.537.328-4.746.655-6.628.901-4.173.572-8.019 1.472-11.292 2.781-3.355 1.31-5.973 3.11-7.855 5.401-1.964 2.291-2.864 5.318-2.864 8.918 0 5.237 1.882 9.164 5.728 11.782 3.682 2.782 8.51 4.091 14.401 4.091Zm64.643 18.328V71.839h27.412v19.965h1.227c2.21-6.955 5.974-12.274 11.292-16.038 5.319-3.763 11.456-5.645 18.329-5.645 1.555 0 3.355.082 5.237.163 1.964.164 3.601.328 4.91.573v25.938c-1.227-.41-3.109-.819-5.646-1.146a58.814 58.814 0 0 0-7.446-.49c-5.155 0-9.738 1.145-13.829 3.354-4.091 2.209-7.282 5.236-9.655 9.164-2.373 3.927-3.519 8.427-3.519 13.5v70.448h-28.312ZM222.077 39.192l-8.019 125.923L137.387 0l84.69 39.192Zm-53.105 162.825-57.933 33.056-57.934-33.056 11.783-28.556h92.301l11.783 28.556ZM111.039 62.675l30.357 73.803H80.681l30.358-73.803ZM7.937 165.115 0 39.192 84.69 0 7.937 165.115Z"

/>

</g>

<defs>

<radialGradient

id="c"

cx="0"

cy="0"

r="1"

gradientTransform="rotate(118.122 171.182 60.81) scale(205.794)"

gradientUnits="userSpaceOnUse"

>

<stop stop-color="#FF41F8" />

<stop offset=".707" stop-color="#FF41F8" stop-opacity=".5" />

<stop offset="1" stop-color="#FF41F8" stop-opacity="0" />

</radialGradient>

<linearGradient

id="b"

x1="0"

x2="982"

y1="192"

y2="192"

gradientUnits="userSpaceOnUse"

>

<stop stop-color="#F0060B" />

<stop offset="0" stop-color="#F0070C" />

<stop offset=".526" stop-color="#CC26D5" />

<stop offset="1" stop-color="#7702FF" />

</linearGradient>

<clipPath id="a"><path fill="#fff" d="M0 0h982v239H0z" /></clipPath>

</defs>

</svg>

<h1>Hello, {{ title }}</h1>

<p>Congratulations! Your app is running. ????</p>

</div>

<div class="divider" role="separator" aria-label="Divider"></div>

<div class="right-side">

<div class="pill-group">

@for (item of [

{ title: 'Explore the Docs', link: 'https://angular.dev' },

{ title: 'Learn with Tutorials', link: 'https://angular.dev/tutorials' },

{ title: 'CLI Docs', link: 'https://angular.dev/tools/cli' },

{ title: 'Angular Language Service', link: 'https://angular.dev/tools/language-service' },

{ title: 'Angular DevTools', link: 'https://angular.dev/tools/devtools' },

]; track item.title) {

<a

class="pill"

href="{{ item.link }}"

target="\_blank"

rel="noopener"

>

<span>{{ item.title }}</span>

<svg

xmlns="http://www.w3.org/2000/svg"

height="14"

viewBox="0 -960 960 960"

width="14"

fill="currentColor"

>

<path

d="M200-120q-33 0-56.5-23.5T120-200v-560q0-33 23.5-56.5T200-840h280v80H200v560h560v-280h80v280q0 33-23.5 56.5T760-120H200Zm188-212-56-56 372-372H560v-80h280v280h-80v-144L388-332Z"

/>

</svg>

</a>

}

</div>

<div class="social-links">

<a

href="https://github.com/angular/angular"

aria-label="Github"

target="\_blank"

rel="noopener"

>

<svg

width="25"

height="24"

viewBox="0 0 25 24"

fill="none"

xmlns="http://www.w3.org/2000/svg"

alt="Github"

>

<path

d="M12.3047 0C5.50634 0 0 5.50942 0 12.3047C0 17.7423 3.52529 22.3535 8.41332 23.9787C9.02856 24.0946 9.25414 23.7142 9.25414 23.3871C9.25414 23.0949 9.24389 22.3207 9.23876 21.2953C5.81601 22.0377 5.09414 19.6444 5.09414 19.6444C4.53427 18.2243 3.72524 17.8449 3.72524 17.8449C2.61064 17.082 3.81137 17.0973 3.81137 17.0973C5.04697 17.1835 5.69604 18.3647 5.69604 18.3647C6.79321 20.2463 8.57636 19.7029 9.27978 19.3881C9.39052 18.5924 9.70736 18.0499 10.0591 17.7423C7.32641 17.4347 4.45429 16.3765 4.45429 11.6618C4.45429 10.3185 4.9311 9.22133 5.72065 8.36C5.58222 8.04931 5.16694 6.79833 5.82831 5.10337C5.82831 5.10337 6.85883 4.77319 9.2121 6.36459C10.1965 6.09082 11.2424 5.95546 12.2883 5.94931C13.3342 5.95546 14.3801 6.09082 15.3644 6.36459C17.7023 4.77319 18.7328 5.10337 18.7328 5.10337C19.3942 6.79833 18.9789 8.04931 18.8559 8.36C19.6403 9.22133 20.1171 10.3185 20.1171 11.6618C20.1171 16.3888 17.2409 17.4296 14.5031 17.7321C14.9338 18.1012 15.3337 18.8559 15.3337 20.0084C15.3337 21.6552 15.3183 22.978 15.3183 23.3779C15.3183 23.7009 15.5336 24.0854 16.1642 23.9623C21.0871 22.3484 24.6094 17.7341 24.6094 12.3047C24.6094 5.50942 19.0999 0 12.3047 0Z"

/>

</svg>

</a>

<a

href="https://twitter.com/angular"

aria-label="Twitter"

target="\_blank"

rel="noopener"

>

<svg

width="25"

height="20"

viewBox="0 0 25 20"

fill="none"

xmlns="http://www.w3.org/2000/svg"

alt="Twitter"

>

<path

d="M8.04524 20C17.3335 20 22.4138 12.3047 22.4138 5.63144C22.4138 5.41287 22.4138 5.19529 22.399 4.97869C23.3874 4.26381 24.2405 3.37867 24.9185 2.3647C23.9969 2.77329 23.0192 3.04112 22.018 3.15923C23.0723 2.52818 23.8613 1.53552 24.2382 0.366057C23.2469 0.954335 22.1624 1.36889 21.0315 1.59182C20.2701 0.782212 19.2631 0.246107 18.1663 0.0664704C17.0695 -0.113166 15.9441 0.0736804 14.9642 0.598096C13.9843 1.12251 13.2046 1.95526 12.7457 2.96748C12.2868 3.9797 12.1742 5.11495 12.4255 6.19756C10.4178 6.09685 8.45366 5.57507 6.66064 4.66609C4.86763 3.75712 3.28579 2.48127 2.01781 0.921344C1.37203 2.03306 1.17424 3.34911 1.46472 4.60154C1.75519 5.85397 2.51208 6.9486 3.58128 7.66257C2.77759 7.63903 1.9914 7.42221 1.28924 7.03049V7.09449C1.28956 8.26041 1.69316 9.39034 2.4316 10.2926C3.17003 11.1949 4.19783 11.8139 5.34067 12.0448C4.59721 12.2476 3.81715 12.2772 3.06045 12.1315C3.38327 13.1348 4.01156 14.0122 4.85746 14.641C5.70337 15.2698 6.72461 15.6185 7.77842 15.6384C6.73139 16.4614 5.53237 17.0699 4.24995 17.4291C2.96753 17.7882 1.62687 17.891 0.304688 17.7316C2.61411 19.2136 5.30121 19.9997 8.04524 19.9961"

/>

</svg>

</a>

<a

href="https://www.youtube.com/channel/UCbn1OgGei-DV7aSRo\_HaAiw"

aria-label="Youtube"

target="\_blank"

rel="noopener"

>

<svg

width="29"

height="20"

viewBox="0 0 29 20"

fill="none"

xmlns="http://www.w3.org/2000/svg"

alt="Youtube"

>

<path

fill-rule="evenodd"

clip-rule="evenodd"

d="M27.4896 1.52422C27.9301 1.96749 28.2463 2.51866 28.4068 3.12258C29.0004 5.35161 29.0004 10 29.0004 10C29.0004 10 29.0004 14.6484 28.4068 16.8774C28.2463 17.4813 27.9301 18.0325 27.4896 18.4758C27.0492 18.9191 26.5 19.2389 25.8972 19.4032C23.6778 20 14.8068 20 14.8068 20C14.8068 20 5.93586 20 3.71651 19.4032C3.11363 19.2389 2.56449 18.9191 2.12405 18.4758C1.68361 18.0325 1.36732 17.4813 1.20683 16.8774C0.613281 14.6484 0.613281 10 0.613281 10C0.613281 10 0.613281 5.35161 1.20683 3.12258C1.36732 2.51866 1.68361 1.96749 2.12405 1.52422C2.56449 1.08095 3.11363 0.76113 3.71651 0.596774C5.93586 0 14.8068 0 14.8068 0C14.8068 0 23.6778 0 25.8972 0.596774C26.5 0.76113 27.0492 1.08095 27.4896 1.52422ZM19.3229 10L11.9036 5.77905V14.221L19.3229 10Z"

/>

</svg>

</a>

</div>

</div>

</div>

</main>

<!-- \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* \* The content above \* \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* is only a placeholder \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* and can be replaced. \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* -->

<!-- \* \* \* \* \* \* \* \* \* \* End of Placeholder \* \* \* \* \* \* \* \* \* \* \* \* -->

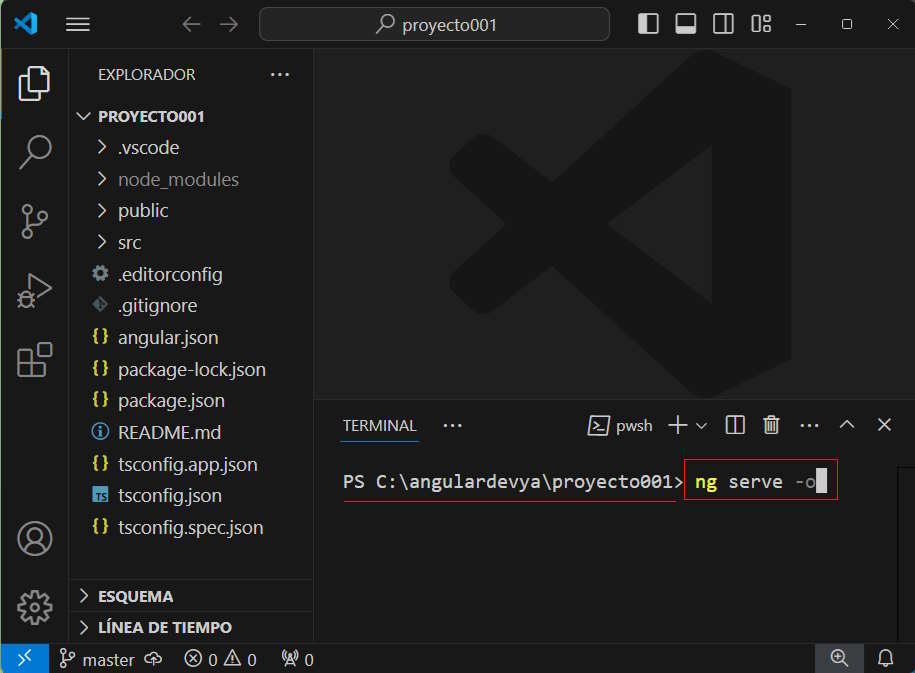
<!-- \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* -->

<router-outlet />

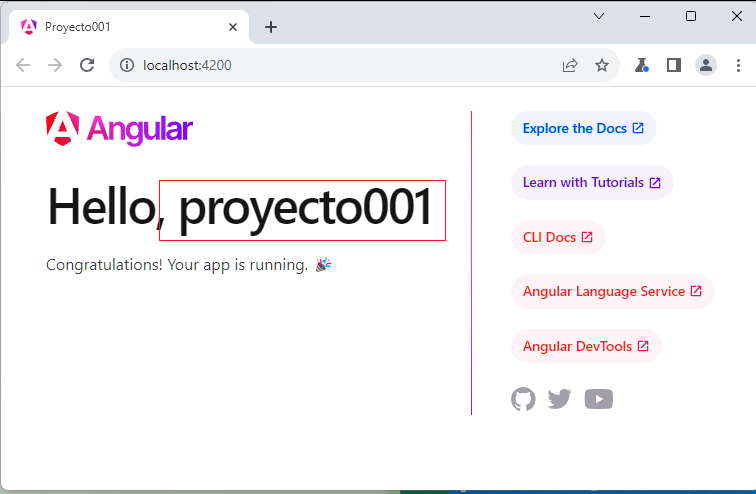
Analizaremos ahora de este trozo de HTML donde aparece el atributo 'title' de la componente:

<h1>Hello, {{ title }}</h1>

Cuando ejecutamos nuestra aplicación desde la línea de comandos de Node.js:



En el navegador aparece el contenido de la propiedad 'title':



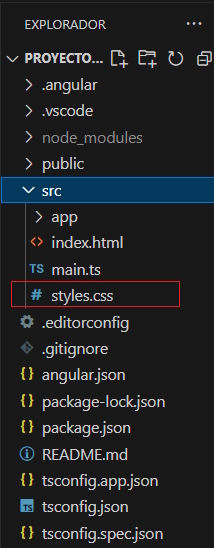
Podemos ver que aparece el string 'proyecto001' y no {{ title }}:

title = 'proyecto001';

Este concepto de sustitución se llama interpolación y lo veremos en forma más profunda en el concepto siguiente.

Otro archivo que se asocia a la componente 'AppComponent' es 'app.component.css' donde se almacenan todos los estilos que se van a aplicar solo a dicha componente, es decir que quedarán encapsulados en la componente 'AppComponent'.

En la carpeta src del proyecto hay un archivo llamado 'styles.css' donde podemos definir estilos que se aplicarán en forma global a todas las componentes de nuestra aplicación:



Ya hemos nombrado los tres archivos fundamentales que definen toda componente:

app.component.ts

app.component.html

app.component.css

Queda uno llamado 'app.component.spec.ts' que tiene por objetivo definir código de testing para medir el correcto funcionamiento de la componente (dejaremos para más adelante este concepto)

Podemos acotar que en versiones anteriores a la 17 de Angular se creaba un archivo fundamental de módulos, pero ahora podemos crear aplicaciones sin la obligatoriedad de insertarlos en módulos.

Hay muchos más archivos y carpetas en el proyecto que nos crea Angular CLI pero iremos viendo su objetivo a medida que avancemos en el curso.

1. Interpolación en los archivos HTML de Angular

Una de las características fundamentales en Angular es separar la vista del modelo de datos. En el modelo de datos tenemos las variables y en la vista implementamos como se muestran dichos datos.

Modificaremos el proyecto001 para ver este concepto de interpolación.

Abriremos el archivo que tiene la clase AppComponent (app.component.ts) y lo modificaremos con el siguiente código:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

nombre = 'Rodriguez Pablo';

edad = 40;

email = 'rpablo@gmail.com';

sueldos = [1700, 1600, 1900];

activo = true;

esActivo() {

if (this.activo)

return 'Trabajador Activo';

else

return 'Trabajador Inactivo';

}

ultimos3Sueldos() {

let suma = 0;

for (let x = 0; x < this.sueldos.length; x++)

suma += this.sueldos[x];

return suma;

}

}

La clase 'AppComponent' representa los datos de un empleado. Definimos e inicializamos 5 propiedades:

nombre = 'Rodriguez Pablo';

edad = 40;

email = 'rpablo@gmail.com';

sueldos = [1700, 1600, 1900];

activo = true;

Definimos dos métodos, en el primero según el valor que almacena la propiedad 'activo' retornamos un string que informa si es un empleado activo o inactivo:

esActivo() {

if (this.activo)

return 'Trabajador Activo';

else

return 'Trabajador Inactivo';

}

El segundo método retorna la suma de sus últimos 3 meses de trabajo que se almacenan en la propiedad 'sueldos':

ultimos3Sueldos() {

let suma=0;

for(let x=0; x<this.sueldos.length; x++)

suma+=this.sueldos[x];

return suma;

}

Veamos ahora el archivo html que muestra los datos, esto se encuentra en 'app.component.html':

<div>

<p>Nombre del Empleado:{{nombre}}</p>

<p>Edad:{{edad}}</p>

<p>Los últimos tres sueldos son: {{sueldos[0]}}, {{sueldos[1]}} y {{sueldos[2]}}</p>

<p>En los últimos 3 meses ha ganado: {{ultimos3Sueldos()}}</p>

<p>{{esActivo()}}</p>

</div>

<router-outlet />

Para acceder a las propiedades del objeto dentro del template del HTML debemos disponer dos llaves abiertas y cerradas y dentro el nombre de la propiedad:

<p>Nombre del Empleado:{{nombre}}</p>

Cuando se tratan de vectores la primer forma que podemos acceder es mediante un subíndice:

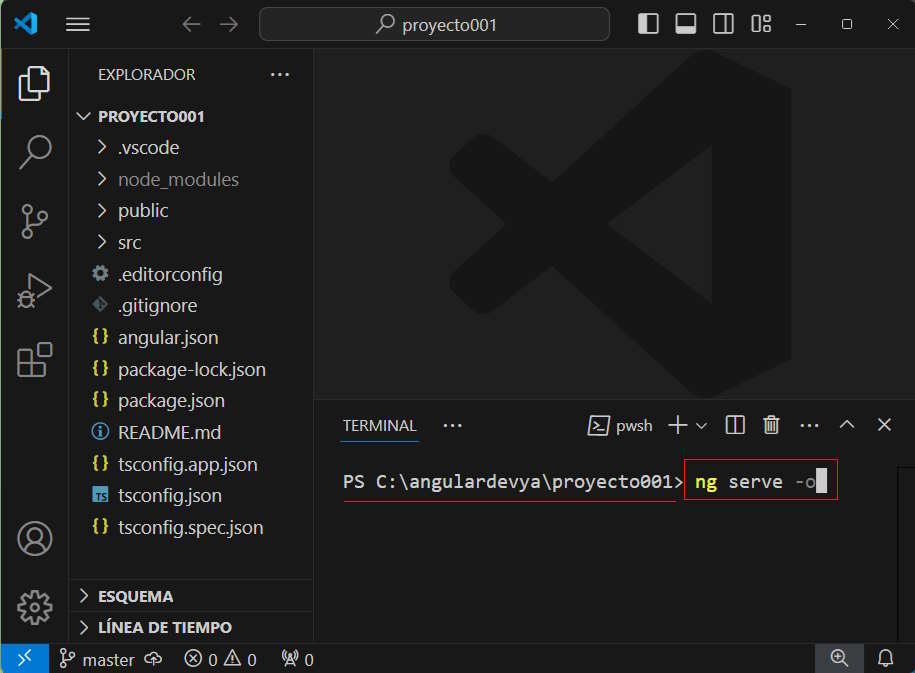
<p>Los últimos tres sueldos son: {{sueldos[0]}}, {{sueldos[1]}} y {{sueldos[2]}}</p>

Finalmente podemos llamar a métodos que tiene por objetivo consultar el valor de propiedades:

<p>En los últimos 3 meses ha ganado: {{ultimos3Sueldos()}}</p>

<p>{{esActivo()}}</p>

Cuando ejecutamos nuestra aplicación desde la línea de comandos de Node.js:



En el navegador aparece el contenido de la vista pero con los valores sustituidos donde dispusimos las llaves {{}}:



En principio podríamos decir que si los datos son siempre los mismos no tiene sentido definir propiedades en la clase y sustituirlos luego en el HTML, pero luego veremos que las propiedades las vamos a cargar mediante una petición a un servidor web, en esas circunstancias veremos la potencia que tiene modificar las propiedades y luego en forma inmediata se modifica la vista.

Acotaciones

* Dentro de las dos llaves abiertas y cerradas Angular nos permite efectuar una operación:

<p>En los últimos 3 meses ha ganado: {{sueldos[0]+sueldos[1]+sueldos[2]}}</p>

Primero se opera la expresión dispuesta dentro de las llaves previo a mostrarla.

Otro ejemplo:

<p>El empleado dentro de 5 años tendrá:{{edad+5}}</p>

* Podemos utilizar la interpolación como valor en propiedades de elementos HTML. Si en la clase tenemos definida la propiedad:
* sitio='http://www.google.com';

Luego en la vista podemos interpolar la propiedad 'url' del elemento 'a' con la siguiente sintaxis:

<p>Puede visitar el sitio ingresando <a href="{{sitio}}">aquí</a></p>

1. Sintaxis de Template para estructuras condicionales y repetitivas: @if / @else - @for - @switch/@case/@default

Angular usa @if para expresar visualizaciones condicionales en plantillas.  
La plantilla o template @if nos permiten condicionar si se deben agregar o no bloques de código.

La plantilla @for nos permite generar muchos elementos HTML repetidos a partir del recorrido de un arreglo de datos.

Para analizar con un ejemplo estas plantillas procederemos nuevamente a modificar el proyecto001.

En el archivo 'app.component.ts' procedemos a codificar la clase AppComponent con la definición de:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

nombre = 'Rodriguez Pablo';

edad = 40;

articulos = [{

codigo: 1,

descripcion: 'naranjas',

precio: 540

}, {

codigo: 2,

descripcion: 'manzanas',

precio: 900

}, {

codigo: 3,

descripcion: 'peras',

precio: 490

}];

generarNumero() {

return Math.floor(Math.random() \* 3) + 1;

}

}

Hemos definido las propiedades nombre, edad y articulos:

nombre = 'Rodriguez Pablo';

edad = 40;

articulos = [{

codigo: 1,

descripcion: 'naranjas',

precio: 540

},{

codigo: 2,

descripcion: 'manzanas',

precio: 900

},{

codigo: 3,

descripcion: 'peras',

precio: 490

}];

Por otro lado un método que retorna un valor aleatorio comprendido entre 1 y 3:

generarNumero() {

return Math.floor(Math.random() \* 3) + 1;

}

Ahora procedemos a modificar el archivo app.component.html:

<div>

<h1>Empleado</h1>

<p>Nombre del Empleado:{{nombre}}</p>

<p>Edad:{{edad}}</p>

@if (edad>=18) {

<p>Es mayor de edad.</p>

} @else {

<p>Es menor de edad.</p>

}

<h1>Listado de articulos</h1>

<table>

@for(articulo of articulos; track articulo.codigo) {

<tr>

<td>{{articulo.codigo}}</td>

<td>{{articulo.descripcion}}</td>

<td>{{articulo.precio}}</td>

</tr>

}

</table>

<h1>Numero aleatorio entre 1 y 3</h1>

@switch (generarNumero()) {

@case (1) {

<p>Uno</p>

}

@case (2) {

<p>Dos</p>

}

@case (3) {

<p>Tres</p>

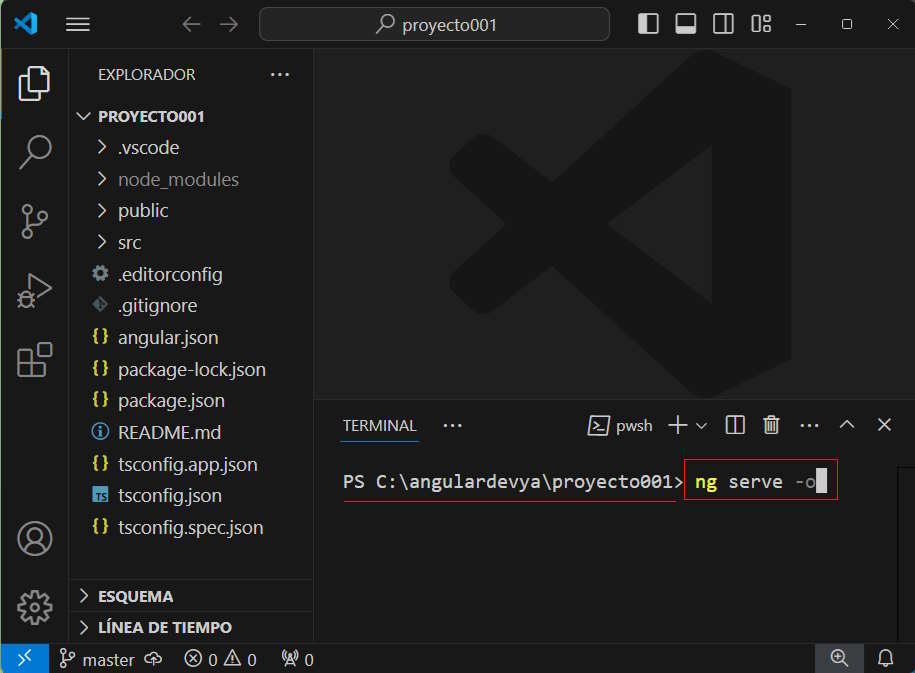
}

}

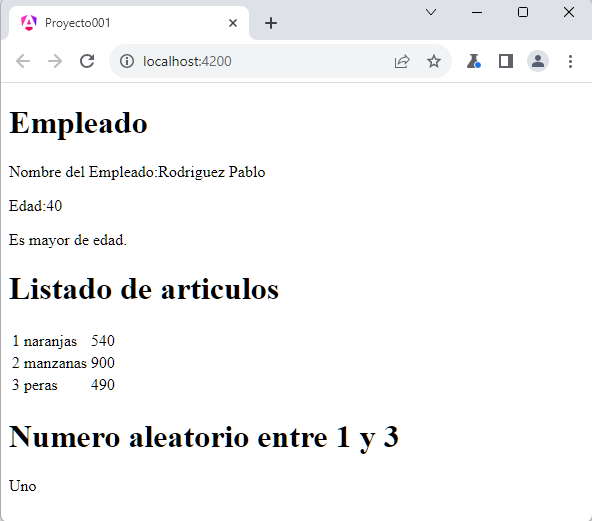
</div>

<router-outlet />

Ejecutemos nuestra aplicación desde la línea de comandos de Node.js:



En el navegador aparece el siguiente contenido:



La instrucción @if verifica la condición, en el caso de verificarse verdadero se inserta el bloque contenido entre las llaves:

@if (edad>=18) {

<p>Es mayor de edad.</p>

} @else {

<p>Es menor de edad.</p>

}

Luego si la condición se verifica falsa se ejecuta el bloque definido en el @else. Podemos probar cambiando la edad por un valor menor a 18.

La instrucción @for nos genera posiblemente muchos elementos HTML repetidos, en este ejemplo una serie de filas de una tabla HTML:

@for(articulo of articulos; track articulo.codigo) {

<tr>

<td>{{articulo.codigo}}</td>

<td>{{articulo.descripcion}}</td>

<td>{{articulo.precio}}</td>

</tr>

}

En cada repetición en la variable 'articulo' se almacena un objeto del arreglo 'articulos'. De esta forma podemos mostrar los datos del objeto respectivo.

Es importante agregar la sentencia track al @for indicando un valor único en cada vuelta del @for (es común utilizar un id o clave)

Por último también disponemos la instrucción @switch donde llamamos al método 'generarNumero' y según el valor retornado entrará en el @case respectivo:

@switch (generarNumero()) {

@case (1) {

<p>Uno</p>

}

@case (2) {

<p>Dos</p>

}

@case (3) {

<p>Tres</p>

}

}

1. Captura de eventos

Otra actividad muy común en una aplicación es la captura de eventos. La presión de un botón, la presión de una tecla, el desplazamiento de la flecha del mouse etc. son eventos que podemos capturar.

El evento más común que podemos encontrar en cualquier aplicación es la presión de un botón. Modificaremos nuevamente el proyecto001 para que la componente AppComponent muestre un etiqueta con un número 0 y luego dos botones que permitan incrementar o decrementar en uno el contenido de la etiqueta.

Nuevamente debemos modificar el archivo 'app.component.ts' con el siguiente código:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

contador = 1;

incrementar() {

this.contador++;

}

decrementar() {

this.contador--;

}

}

Definimos en la clase la propiedad 'contador' y lo iniciamos con el valor '1':

export class AppComponent {

contador = 1;

Luego otros dos métodos de la clase AppComponent, que serán llamados al presionar alguno de los botones, incrementan en uno o decrementan en uno el valor almacenado en la propiedad contador:

incrementar() {

this.contador++;

}

decrementar() {

this.contador--;

}

Recordar que las propiedades dentro de los métodos debemos anteceder la palabra clave 'this'

El segundo archivo donde se encuentra la vista de la componente es app.component.html:

<div>

<p>{{contador}}</p>

<button (click)="incrementar()">Sumar 1</button>

<button (click)="decrementar()">Restar 1</button>

</div>

<router-outlet />

Como ya conocemos mostramos el contenido de la propiedad contador mediante interpolación de string:

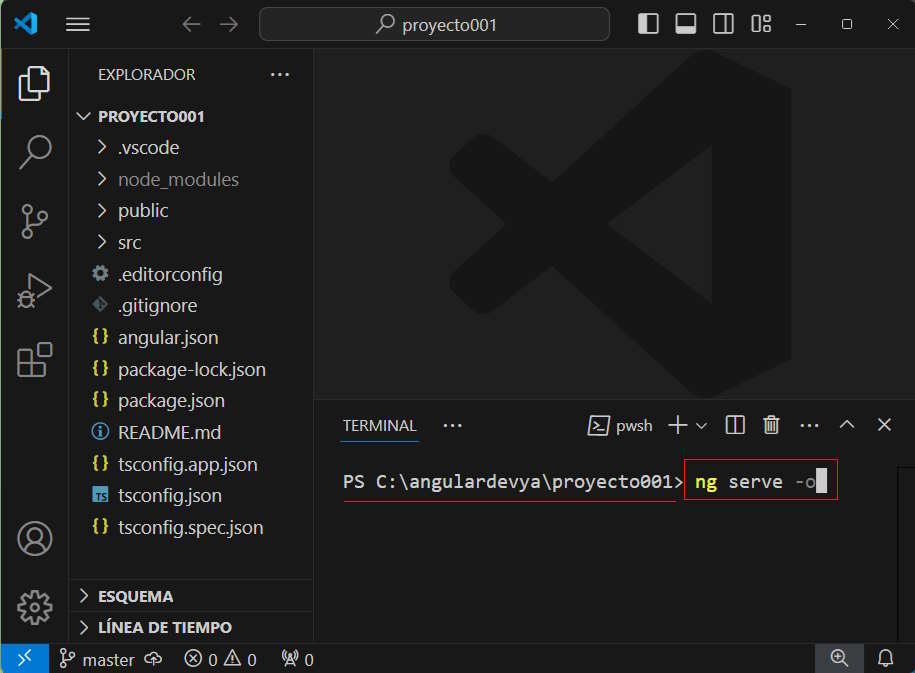
<p>{{contador}}</p>

Luego definimos dos elementos HTML de tipo 'button' y definimos los eventos click (deben ir entre paréntesis los nombres de los eventos) y luego entre comillas el nombre del método que se llama:

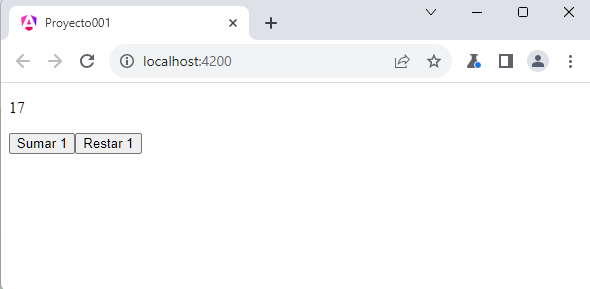
<button (click)="incrementar()">Sumar 1</button>

<button (click)="decrementar()">Restar 1</button>

Ejecutemos nuestra aplicación desde la línea de comandos de Node.js:



En el navegador aparece la siguiente interfaz:



Cuando se presiona el botón 'Sumar 1' se llama el método 'incrementar()', en dicho método si recordamos se modifica el contenido de la propiedad 'contador':

incrementar() {

this.contador++;

}

Lo más importante notar que Angular detecta cuando se modifican valores almacenados en propiedades y automáticamente se encarga de actualizar la interfaz visual sin tener que llamar a algún método.

Este concepto se conoce como 'binding' en una dirección (cambio en atributos de la clase se actualizan en la vista)

1. Enlace de propiedades (Property Binding)

El property binding es una herramienta poderosa en Angular que nos permite mantener sincronizadas las propiedades de un componente con las propiedades de elementos del DOM, facilitando así la creación de aplicaciones dinámicas e interactivas.

Vimos que para la captura de eventos en los elementos HTML colocamos entre paréntesis el evento y le asociamos un método, para el enlace de propiedades debemos colocar entre corchetes el nombre de la propiedad y asignarle un atributo definido en la clase.

Modifiquemos nuevamente el archivo 'app.component.ts' con el siguiente código:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

nombre = ''

fijarNombre1() {

this.nombre = 'Juan';

}

fijarNombre2() {

this.nombre = 'Ana';

}

}

Definimos un atributo llamado nombre, el cual se modifica en dos métodos.

El segundo archivo donde se encuentra la vista de la componente y donde definimos el property binding es app.component.html:

<div>

<input type="button" (click)="fijarNombre1()" value="Juan">

<input type="button" (click)="fijarNombre2()" value="Ana">

</div>

<input type="text" [value]="nombre">

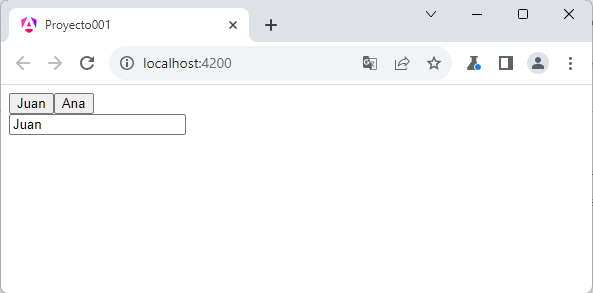
<router-outlet />

Debemos disponer entre corchetes el nombre de la propiedad y asignarle el nombre del atributo definido en la clase:

<input type="text" [value]="nombre">

Cuando el operador presione cualquiera de los dos botones, al modificarse el atributo "nombre" en la clase, automáticamente se actualiza la vista con el valor asignado al atributo.

Si ejecutamos la aplicación y presionamos el primer botón:



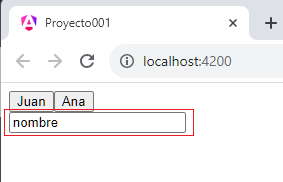
Tener cuidado con la importancia de los corchetes que envuelven la propiedad:

<input type="text" [value]="nombre">

Si no disponemos los corchetes, recordemos que el navegador va a mostrar la cadena "nombre" dentro del control de entrada de datos:

<input type="text" value="nombre">

Y tenemos como resultado en la página:



1. Componentes: creación

Hasta ahora hemos desarrollado toda la lógica en el componente que se crea por defecto al crear un proyecto con Angular CLI.

La realidad es que en proyectos de mediano y gran tamaño no podemos disponer toda la lógica en un único componente.

Los componentes son una de las características fundamentales de Angular. Ayudan a extender las características básicas de las etiquetas HTML y encapsular código.

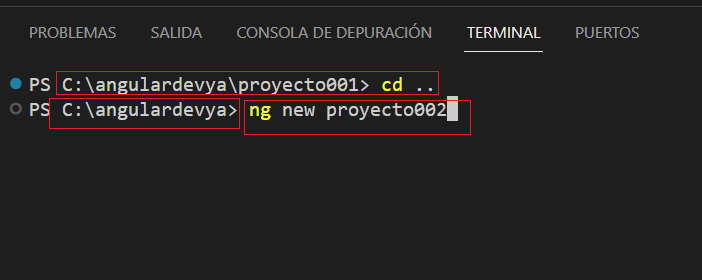
Para crear otros componentes la herramienta Angular CLI nos provee la posibilidad de crearlas desde la línea de comandos de Node.js

Problema

Implementar una aplicación que muestre tres dados. Crear una componente 'dado' además de la componente que crea por defecto Angular CLI

Seguir los siguientes pasos para implementar el proyecto002:

1. Desde la línea de comandos de Node.js o terminal de VS Code procedemos a crear el segundo proyecto (cuidado de crearlo desde la carpeta c:\angujardevya\):



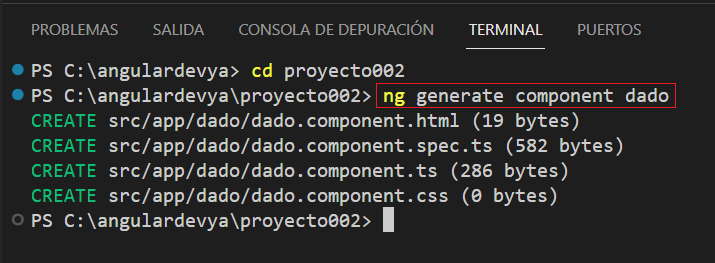
c:\angujardevya>ng new proyecto002

Primero descendemos a la carpeta proyecto002 y nuevamente desde la línea de comandos procedemos a crear la componente 'dado' escribiendo:

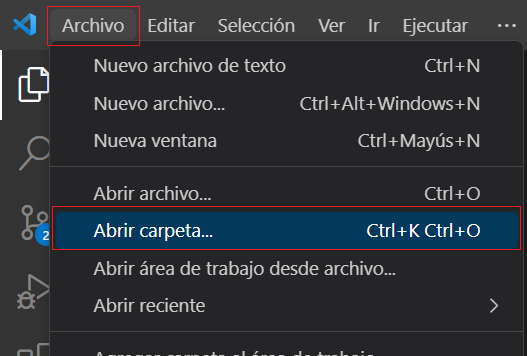
c:\angularya> cd proyecto002

c:\angularya\proyecto002> ng generate component dado

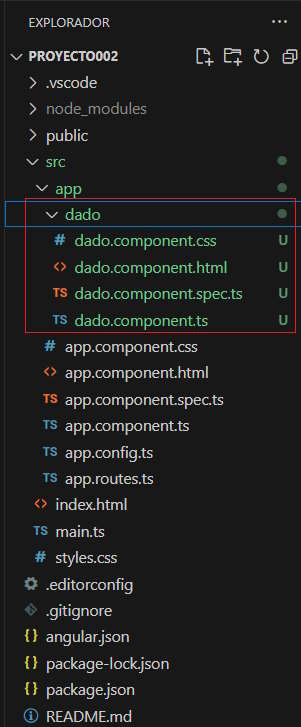
Al ejecutar este comando se crean 4 archivos:



Ahora desde el VSCode ya podemos activar la carpeta donde se encuentra creado el segundo proyecto:



Ahora podemos comprobar que dentro de la carpeta 'app' se crea una carpeta llamada 'dado' y dentro de ella se localizan los cuatro archivos creados:



En nuestro tercer paso vamos a implementar la vista de la componente 'dado' y su modelo. Abrimos el archivo 'dado.component.ts' y codificamos:

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

@Component({

selector: 'app-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

export class DadoComponent {

valor: number = Math.floor(Math.random() \* 6) + 1;

}

La propiedad valor se inicializa con un valor aleatorio comprendido entre 1 y 6.

Codificamos ahora el archivo 'dado.component.html':

<div class="forma">

{{valor}}

</div>

Como podemos ver solo mostramos el valor almacenado en la propiedad 'valor' definido en el modelo.

Para definir la hoja de estilo del 'dado' abrimos el archivo 'dado.component.css' y codificamos:

.forma {

width: 5rem;

height: 5rem;

font-size: 3rem;

color:white;

background-color: black;

border-radius: 1rem;

display: inline-flex;

justify-content: center;

align-items: center;

margin:10px;

}

Finamente nos falta definir tres objetos de nuestra clase 'DadoComponent', si volvemos a ver el archivo 'dado.component.ts' podemos identificar en la llamada a @Component que tiene una propiedad llamada 'selector' con el valor 'app-dado':

@Component({

selector: 'app-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

Este es el selector que debemos utilizar para definir objetos de la clase DadoComponent en las vistas.

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de tres dados:

<div style="text-align:center">

<app-dado></app-dado>

<app-dado></app-dado>

<app-dado></app-dado>

</div>

<router-outlet />

Todavía podemos ver que nos muestra un error como que no existe la etiqueta 'app-dado', esto debido que la debemos importar a la clase en el archivo 'app.component.ts'.

Debemos importar la clase e indicar que la utilizaremos:

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

import { RouterOutlet } from '@angular/router';

**import { DadoComponent } from './dado/dado.component';**

@Component({

selector: 'app-root',

imports: [RouterOutlet, **DadoComponent**],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

title = 'proyecto002';

}

Es decir que importamos la componente del dado:

import { DadoComponent } from './dado/dado.component';

Y también indicamos el nombre de la clase en la propiedad 'imports':

imports: [RouterOutlet, DadoComponent]

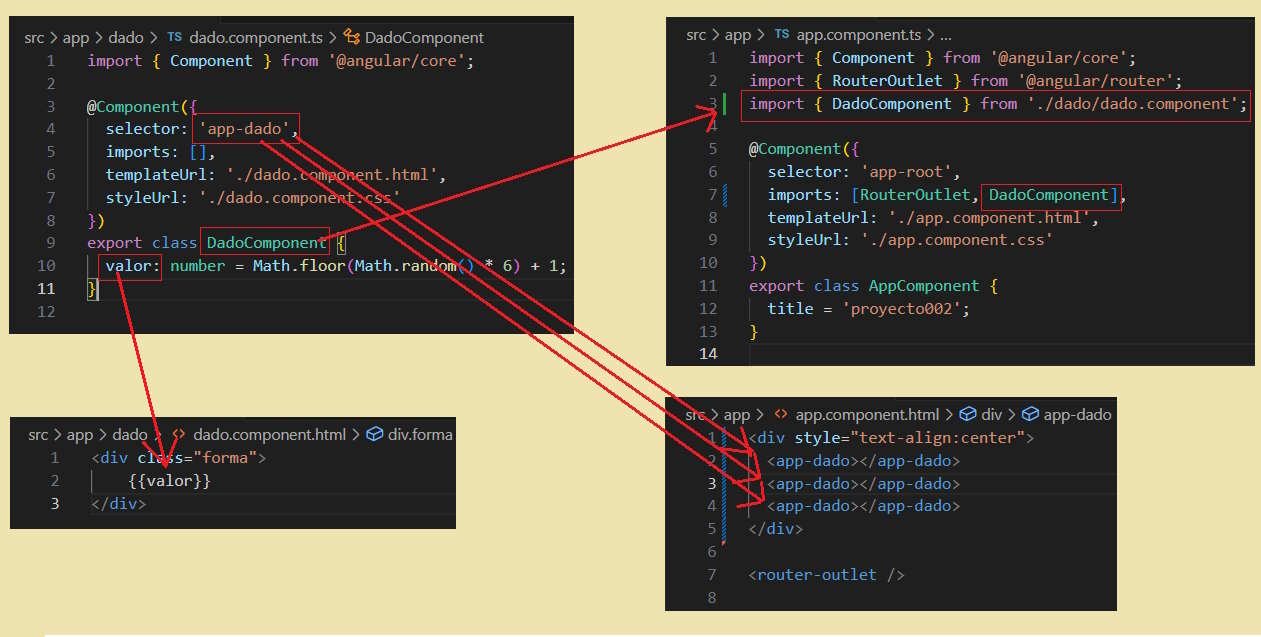
Si ejecutamos ahora el proyecto:

ng serve -o

Podemos ver que tenemos los tres dados en pantalla:



Es importante notar la relación de colaboración entre la clase 'DadoComponent' y la clase 'AppComponent':



La clase AppComponent se crea en forma automática cuando creamos un proyecto en Angular utilizando la herramienta 'Angular CLI'.

Acotaciones

La división de un proyecto en componentes en Angular nos permite crear piezas independientes y reutilizables.

Siempre debe haber una primer componente donde arranca la aplicación, si utilizamos la herramienta Angular CLI se llama 'AppComponent'. Luego podemos crear otras componentes como en nuestro caso de 'DadoComponent'.

Tenemos que toda componente tiene un nombre de clase, por ejemplo 'DadoComponent' y luego un nombre de selector definida para dicha componente 'app-dado'. En las vistas para definir objetos de una determinada componente debemos hacer referencia al nombre del selector:

<app-dado></app-dado>

La componente 'AppComponent' si vemos su nombre de selector es 'app-root', luego si queremos ver donde se crea un objeto de este tipo de selector debemos abrir el archivo 'index.html' que lo generó automáticamente Angular CLI:

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Proyecto002</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

</html>

Como podemos ver en el 'body' se define una etiqueta de tipo 'app-root':

<body>

<app-root></app-root>

</body>

Gracias a esto siempre se crea un objeto de la clase AppComponent. Si borramos ésta etiqueta y corremos la aplicación luego tendremos una página web vacía.

1. Componentes: pasar datos del componente padre al componente hijo

Vimos en el concepto anterior que una aplicación con Angular está conformada por componentes. Siempre hay un componente padre y esta puede tener uno o más componentes hijos, a su vez los componentes hijos pueden tener componentes hijos de ellos y así sucesivamente.

En este concepto veremos una técnica para pasar datos del componente padre al componente hijo.

Podemos pasar datos a un componente en el momento que definimos una etiqueta de la misma:

<app-dado valor="3"></app-dado>

Cuando declaramos la etiqueta app-dado definimos una propiedad llamada 'valor' y le pasamos el dato a dicho componente. Es idéntico a lo que hacemos a cuando definimos etiquetas HTML con sus propiedades.

Problema

Implementar una aplicación que muestre tres dados (crear un componente llamado 'dado') y un botón. Cuando se presione el botón generar tres valores aleatorios y pasarlos a los componentes respectivos para que se muestren. La clase principal debe mostrar un mensaje que el usuario ganó si los tres dados tienen el mismo valor.

Repetiremos los mismos pasos del concepto anterior para practicar la creación de un nuevo componente y veremos lo nuevo de recibir datos en la componente creada.

1. Desde la línea de comandos de Node.js procedemos a crear el proyecto003:
2. c:\angulardevya> ng new proyecto003
3. Primero descendemos a la carpeta proyecto003 y nuevamente desde la línea de comandos procedemos a crear el componente 'dado' escribiendo:
4. c:\angulardevya\proyecto003> ng generate component dado

Recordemos que al ejecutar este comando se crean 4 archivos, todos relacionados al componente que acabamos de crear.

Además, dentro de la carpeta 'app' se crea una carpeta llamada 'dado' y dentro de ella se localizan los cuatro archivos creados.

En nuestro tercer paso vamos a implementar la vista de la componente 'dado' y su modelo. Abrimos el archivo 'dado.component.ts' y codificamos:

import { Component, **Input** } from '@angular/core';

@Component({

selector: 'app-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

export class DadoComponent {

**@Input() valor: string = "";**

}

En la clase DadoComponent podemos identificar la sintaxis para definir una propiedad que llega como parámetro de la componente padre:

@Input() valor: string="";

Asignamos un string vacío a la propiedad valor y luego se cargará un valor cuando creemos un objeto de esta clase.

Para definir el decorador @Input() debemos importar la clase Input:

import { Component, **Input** } from '@angular/core';

Codificamos ahora el archivo 'dado.component.html':

<div class="forma">

{{valor}}

</div>

Como podemos ver solo mostramos el valor almacenado en la propiedad 'valor' definido en el modelo (no hay cambios con el ejemplo anterior)

Para definir la hoja de estilo del 'dado' abrimos el archivo 'dado.component.css' y codificamos:

.forma {

width: 5rem;

height: 5rem;

font-size: 3rem;

color:white;

background-color: black;

border-radius: 1rem;

display: inline-flex;

justify-content: center;

align-items: center;

margin:10px;

}

Finamente nos falta definir tres objetos de nuestra clase 'DadoComponent' y pasar los valores que queremos que se muestren, si volvemos a ver el archivo 'dado.component.ts' podemos identificar en la llamada a @Component que tiene una propiedad llamada 'selector' con el valor 'app-dado':

@Component({

selector: 'app-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

Este es el selector que debemos utilizar para definir objetos de la clase DadoComponent en las vistas.

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de tres dados y mediante interpolación pasamos el valor para cada dado:

<div style="text-align:center">

<app-dado valor="{{valor1}}"></app-dado>

<app-dado valor="{{valor2}}"></app-dado>

<app-dado valor="{{valor3}}"></app-dado>

<hr>

<button (click)="tirar()">Tirar</button>

<hr>

<p>Resultado:{{resultado}}</p>

</div>

<router-outlet />

Ahora codificamos la clase AppComponent donde generamos los tres valores aleatorios que mostrarán las componentes, implementamos además el método que captura el click del botón:

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

import { RouterOutlet } from '@angular/router';

import { DadoComponent } from './dado/dado.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, DadoComponent],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

valor1: number = this.retornarAleatorio()

valor2: number = this.retornarAleatorio()

valor3: number = this.retornarAleatorio()

resultado: string = "";

retornarAleatorio() {

return Math.floor(Math.random() \* 6) + 1;

}

tirar() {

this.valor1 = this.retornarAleatorio();

this.valor2 = this.retornarAleatorio();

this.valor3 = this.retornarAleatorio();

if (this.valor1 == this.valor2 && this.valor1 == this.valor3)

this.resultado = 'Ganó';

else

this.resultado = 'Perdió';

}

}

Veamos las partes importantes de la componente 'AppComponent', primero importamos la clase 'DadoComponent':

import { DadoComponent } from './dado/dado.component';

Agregamos a la propiedad imports la referencia a la clase 'DadoComponent':

imports: [RouterOutlet, DadoComponent],

Definimos 4 atributos y los inicializamos a los tres primeros con valores aleatorios y al cuarto con un string vacío:

valor1: number = this.retornarAleatorio()

valor2: number = this.retornarAleatorio()

valor3: number = this.retornarAleatorio()

resultado: string="";

El método tirar, vuelve a generar 3 valores aleatorios para cada dado y según sus valores fija el valor del atributo 'resultado':

tirar() {

this.valor1 = this.retornarAleatorio();

this.valor2 = this.retornarAleatorio();

this.valor3 = this.retornarAleatorio();

if (this.valor1 == this.valor2 && this.valor1 == this.valor3)

this.resultado = 'Ganó';

else

this.resultado = 'Perdió';

}

Recordemos que el archivo HTML mediante interpolación actualiza los valores en forma automática:

<div style="text-align:center">

<app-dado valor="{{valor1}}"></app-dado>

<app-dado valor="{{valor2}}"></app-dado>

<app-dado valor="{{valor3}}"></app-dado>

<hr>

<button (click)="tirar()">Tirar</button>

<hr>

<p>Resultado:{{resultado}}</p>

</div>

<router-outlet />

Si ejecutamos ahora el proyecto:

ng server -o

Podemos ver que tenemos los tres dados en pantalla, el botón de 'tirar' y el mensaje que se actualiza cada vez que jugamos:



Acotaciones

Dependiendo del problema nos conviene definir propiedades privadas a una componente o definir propiedades que lleguen los datos desde la componente padre. En éste problema como tenemos que controlar si los tres dados tienen el mismo valor es más conveniente que el dado tenga solo la responsabilidad de mostrar un valor y que el control de los tres números se debe hacer en la componente principal 'AppComponent'.

Property binding (enlace de propiedades)

Vimos que podemos modificar el valor de un propiedad en el archivo HTML mediante interpolación:

<app-dado valor="{{valor1}}"></app-dado>

<app-dado valor="{{valor2}}"></app-dado>

<app-dado valor="{{valor3}}"></app-dado>

La segundo forma de resolver el problema en lugar de interpolación es mediante el enlace de propiedades, también llamado 'property binding' que vimos en conceptos previos.

Debemos encerrar entre corchetes el nombre de la propiedad y luego asignar un atributo definido en el modelo de datos:

<app-dado [valor]="valor1"></app-dado>

<app-dado [valor]="valor2"></app-dado>

<app-dado [valor]="valor3"></app-dado>

Con property binding debemos definir en la componente el tipo de dato number:

import { Component, Input } from '@angular/core';

@Component({

selector: 'app-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

export class DadoComponent {

@Input() valor: number=1;

}

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto003/).

A medida que hagamos ejercicios veremos cuando conviene más emplear interpolación o definir property binding.

1. Componentes: disparo de eventos de la componente hija a la componente padre

En el concepto anterior vimos al final como podemos pasar un dato mediante el 'property binding' (la variable 'valor1' se encuentra definida en la clase):

<app-dado [valor]="valor1"></app-dado>

Ahora veremos cómo podemos captura un evento en la componente padre que emite la componente hija:

<app-cronometro[inicio]="15"(multiplo10)="actualizar($event)"></app-cronometro>

En esta componente tenemos una propiedad llamada inicio que le enviamos un dato y capturamos un evento llamado 'multiplo10' que emite la componente app-cronometro.

Problema

Confeccionar una aplicación con dos componentes llamadas 'AppComponent' y 'CronometroComponent'. La componente 'CronometroComponent' muestra un cronómetro que se actualiza cada un segundo, cada vez que su valor es múltiplo de 10 informa a la componente padre de dicha situación informando el segundo actual.

La componente 'AppComponent' define un cronómetro e informa cada vez que el cronómetro tiene un valor múltiplo de 10.

Desde la línea de comandos de Node.js procedemos a crear el proyecto004:

c:\angulardevya> ng new proyecto004

Primero descendemos a la carpeta proyecto004 y nuevamente desde la línea de comandos procedemos a crear la componente 'cronometro' escribiendo:

c:\angulardevya\proyecto004> ng generate component cronometro

Recordemos que al ejecutar este comando se crean 4 archivos.

Además dentro de la carpeta 'app' se crea una carpeta llamada 'cronometro' y dentro de ella se localizan los cuatro archivos creados.

1. En nuestro tercer paso vamos a implementar la vista de la componente 'cronometro' y su modelo. Abrimos el archivo 'cronometro.component.ts' y codificamos:

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-cronometro',

imports: [],

templateUrl: './cronometro.component.html',

styleUrl: './cronometro.component.css'

})

export class CronometroComponent {

segundo = 0;

@Input() inicio: number = 0;

@Output() multiplo10 = new EventEmitter<number>();

ngOnInit() {

this.segundo = this.inicio;

setInterval(() => {

this.segundo++;

if (this.segundo % 10 == 0)

this.multiplo10.emit(this.segundo);

}, 1000);

}

}

En la clase CronometroComponent podemos identificar la sintaxis para definir un evento que dispara un valor de tipo 'number':

@Output() multiplo10 = new EventEmitter<number>();

Para definir los decoradores @Input(), @Output() debemos importar:

import { Component, Input, Output, EventEmitter } from '@angular/core';

También hay que importar la clase EventEmitter.

Aparece algo nuevo que es el método 'ngOnInit' que se ejecuta una vez que se han inicializado todos los atributos. En el mismo, inicializamos el atributo 'segundo' con el valor del decorador 'inicio'.

Mediante la función setInterval, especificamos que cada 1 segundo (1000 milisegundos), se ejecute la arrow function donde incrementamos el atributo 'segundo' y si el mismo es múltiplo de 10 procedemos a llamar a la función de la otra clase.

Codificamos ahora el archivo 'cronometro.component.html':

<div class="cronometro">

{{segundo}} Seg.

</div>

Para definir la hoja de estilo del 'cronometro' abrimos el archivo 'cronometro.component.css' y codificamos:

.cronometro {

width: 8rem;

height: 3rem;

font-size: 2rem;

color:white;

background-color: black;

border-radius: 10px;

display: inline-flex;

justify-content: center;

align-items: center;

margin:10px;

}

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de un cronometro y un mensaje que se muestra mediante interpolación:

<div style="text-align:center">

<h1>Prueba de la componente cronometro</h1>

<app-cronometro [inicio]="15" (multiplo10)="actualizar($event)"></app-cronometro>

<h2>Evento</h2>

<h3>{{mensaje}}</h3>

</div>

<router-outlet />

Es importante entender la sintaxis del evento 'multiplo10' donde se llama al método actualizar.

El $event es una variable especial en Angular que representa el objeto del evento que se está produciendo, en nuestro caso es el valor que se pasa desde el cronómetro del segundo actual:

this.multiplo10.emit(this.segundo);

Ahora codificamos la clase AppComponent donde definimos el método que captura el evento emitido por el cronómetro:

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

import { RouterOutlet } from '@angular/router';

import { CronometroComponent } from './cronometro/cronometro.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, CronometroComponent],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

mensaje = '';

actualizar(t: number) {

this.mensaje = t + '(se actualiza cada 10 segundos)';

}

}

El método 'actualizar' se llama cuando se dispara el evento 'multiplo10':

<app-cronometro [inicio]="15" (multiplo10)="actualizar($event)"></app-cronometro>

El parámetro 't' del método recibe el valor de la variable especial de Angular llamada $event.

Si ejecutamos ahora el proyecto:

ng server -o

Podemos ver que cada vez que el cronómetro tiene un valor múltiplo de 10 la componente principal actualiza un mensaje gracias al evento que emite la componente 'cronometro':



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto004/).

1. Componentes: llamar a métodos del componente hijo desde el template del padre

Otra forma de comunicarnos del componente padre al componente hijo es la posibilidad de llamar a métodos del componente hijo definiendo una variable en el template HTML donde declaramos el componente hijo.

Para llamar a los métodos la componente debe definir una variable en el template del HTML:

<app-dado #dado1></app-dado>

<button (click)="dado1.tirar()">Tirar</button>

Para definir una variable local le antecedemos el caracter # al nombre. Luego podemos llamar a métodos indicando el nombre de la variable y el método a llamar.

Problema

Crear un componente llamado 'selectornumerico' que le pasemos el mínimo y el máximo valor que puede mostrar. Dentro del componente definir dos botones que puedan incrementar o decrementar en 1 el valor actual.  
En el template HTML de la componente padre definir un selectornumerico y una variable local para poder llamar luego a un método del selectornumerico para fijar cualquier valor.

La componente selectornumerico debe ser algo similar a esta interfaz:



Desde la línea de comandos de Node.js procedemos a crear el proyecto005:

c:\angulardevya> ng new proyecto005

Primero descendemos a la carpeta proyecto005 y nuevamente desde la línea de comandos procedemos a crear la componente 'selectornumerico' escribiendo:

c:\angulardevya\proyecto005> ng generate component selectornumerico

Recordemos que al ejecutar este comando se crean 4 archivos.

Además dentro de la carpeta 'app' se crea una carpeta llamada 'selectornumerico' y dentro de ella se localizan los cuatro archivos creados.

En nuestro tercer paso vamos a implementar la vista de la componente 'selectornumerico' y su modelo. Abrimos el archivo 'selectornumerico.component.ts' y codificamos:

import { Component, Input } from '@angular/core';

@Component({

selector: 'app-selectornumerico',

imports: [],

templateUrl: './selectornumerico.component.html',

styleUrl: './selectornumerico.component.css'

})

export class SelectornumericoComponent {

@Input() minimo: number = 1;

@Input() maximo: number = 1;

actual: number = 1;

ngOnInit() {

this.actual = this.minimo;

}

incrementar() {

if (this.actual < this.maximo)

this.actual++;

}

decrementar() {

if (this.actual > this.minimo)

this.actual--;

}

fijar(v: number) {

if (v >= this.minimo && v <= this.maximo)

this.actual = v;

}

}

En la clase SelectornumericoComponent podemos identificar el método que llamaremos desde el HTML template de la componente AppComponent:

fijar(v: number) {

if (v >= this.minimo && v <= this.maximo)

this.actual = v;

}

Codificamos ahora el archivo 'selectornumerico.component.html':

<div class="selector">

<button (click)="decrementar()">-</button>

<div class="valor">{{actual}}</div>

<button (click)="incrementar()">+</button>

</div>

Para definir la hoja de estilo del 'selectornumerico' abrimos el archivo 'selectornumerico.component.css' y codificamos:

.selector {

display:inline-flex;

margin:0.2rem;

}

.valor {

display:inline-flex;

justify-content: center;

align-items: center;

width: 3rem;

height: 3rem;

background:#ff0;

font-size:2rem;

}

button {

height: 3rem;

}

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de un selectornumerico que define una variable local y tres botones que llaman a partir de esa variable al método definido dentro del selectornumerico:

<div>

<app-selectornumerico [minimo]="1" [maximo]="10" #selector1></app-selectornumerico>

<br>

<button (click)="selector1.fijar(1)">Fijar en 1</button><br>

<button (click)="selector1.fijar(5)">Fijar en 5</button><br>

<button (click)="selector1.fijar(10)">Fijar en 10</button>

</div>

<router-outlet />

En la clase AppComponent nos queda importar la clase 'SelectornumericoComponent' que se encuentra en el archivo 'selectornumerico.component':

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

import { RouterOutlet } from '@angular/router';

import { SelectornumericoComponent } from './selectornumerico/selectornumerico.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, SelectornumericoComponent],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

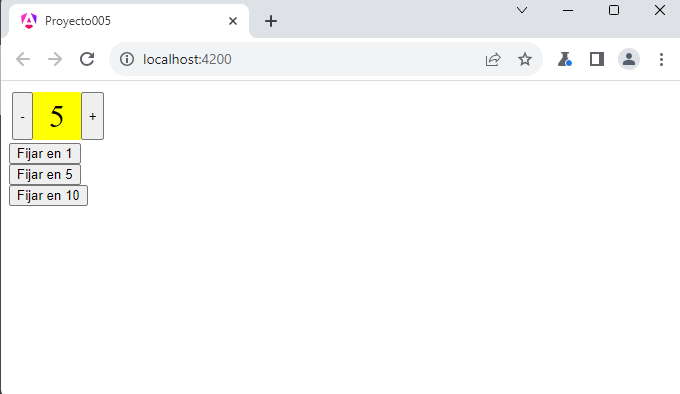
})

export class AppComponent {

}

Si ejecutamos ahora el proyecto:

ng server -o



Es decir que mediante dos botones definidos en la componente AppComponent podemos llamar al método fijar de la clase SelectornumericoComponent gracias a que definimos la variable #selector1:

<app-selectornumerico [minimo]="1" [maximo]="10" #selector1></app-selectornumerico>

<br>

<button (click)="selector1.fijar(1)">Fijar en 1</button><br>

<button (click)="selector1.fijar(5)">Fijar en 5</button><br>

<button (click)="selector1.fijar(10)">Fijar en 10</button>

Lo que debe quedar claro que debemos definir una variable en la etiqueta que define la componente y a partir de esta podemos llamar a métodos o inclusive acceder a propiedades de la componente.

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto005/).

1. Componentes: llamar a métodos del componente hijo desde la clase padre

En muchas situaciones podemos resolver un problema definiendo una variable local en el template HTML del padre y mediante ésta llamar a sus métodos.

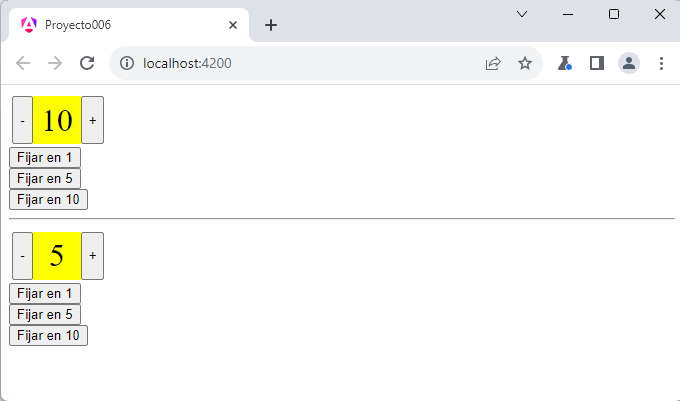
Si necesitamos acceder al componente hijo pero desde la clase (es decir el archivo .ts) tenemos un poco mayor de complejidad que pasaremos a ver en este concepto.

Problema

Crear una componente llamada 'selectornumerico' que le pasemos el mínimo y el máximo valor que puede mostrar. Dentro de la componente definir dos botones que puedan incrementar o decrementar en 1 el valor actual (no modificaremos nada la del ejercicio anterior)

En el template HTML del componente padre definir dos selectores numéricos y tres botones que afectarán a cada selector numérico para fijar los valores 1, 5 y 10.

La aplicación completa se debe mostrar con una interfaz similar a:



Desde la línea de comandos de Node.js procedemos a crear el proyecto006:

c:\angulardevya> ng new proyecto006

Primero descendemos a la carpeta proyecto006 y nuevamente desde la línea de comandos procedemos a crear la componente 'selectornumerico' escribiendo:

c:\angulardevya\proyecto006> ng generate component selectornumerico

En nuestro tercer paso vamos a implementar la vista de la componente 'selectornumerico' y su modelo. Abrimos el archivo 'selectornumerico.component.ts' y codificamos:

import { Component, Input } from '@angular/core';

@Component({

selector: 'app-selectornumerico',

imports: [],

templateUrl: './selectornumerico.component.html',

styleUrl: './selectornumerico.component.css'

})

export class SelectornumericoComponent {

@Input() minimo: number = 1;

@Input() maximo: number = 1;

actual: number = 1;

ngOnInit() {

this.actual = this.minimo;

}

incrementar() {

if (this.actual < this.maximo)

this.actual++;

}

decrementar() {

if (this.actual > this.minimo)

this.actual--;

}

fijar(v: number) {

if (v >= this.minimo && v <= this.maximo)

this.actual = v;

}

}

En la clase SelectornumericoComponent podemos identificar el método que llamaremos desde la clase AppComponent:

fijar(v: number) {

if (v >= this.minimo && v <= this.maximo)

this.actual = v;

}

Codificamos ahora el archivo 'selectornumerico.component.html':

<div class="selector">

<button (click)="decrementar()">-</button>

<div class="valor">{{actual}}</div>

<button (click)="incrementar()">+</button>

</div>

Para definir la hoja de estilo del 'selectornumerico' abrimos el archivo 'selectornumerico.component.css' y codificamos:

.selector {

display:inline-flex;

margin:0.2rem;

}

.valor {

display:inline-flex;

justify-content: center;

align-items: center;

width: 3rem;

height: 3rem;

background:#ff0;

font-size:2rem;

}

button {

height: 3rem;

}

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de dos selectores numericos que definen una variable local y tres botones para cada uno:

<div>

<app-selectornumerico [minimo]="1" [maximo]="10" #selector1></app-selectornumerico>

<br>

<button (click)="fijarSelector1(1)">Fijar en 1</button><br>

<button (click)="fijarSelector1(5)">Fijar en 5</button><br>

<button (click)="fijarSelector1(10)">Fijar en 10</button>

<hr>

<app-selectornumerico [minimo]="1" [maximo]="10" #selector2></app-selectornumerico>

<br>

<button (click)="fijarSelector2(1)">Fijar en 1</button><br>

<button (click)="fijarSelector2(5)">Fijar en 5</button><br>

<button (click)="fijarSelector2(10)">Fijar en 10</button>

</div>

<router-outlet />

La clase AppComponent es la que presenta la mayor cantidad de cambios:

import { Component, ViewChild } from '@angular/core';

import { RouterOutlet } from '@angular/router';

import { SelectornumericoComponent } from './selectornumerico/selectornumerico.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

@ViewChild('selector1') selector1!: SelectornumericoComponent;

@ViewChild('selector2') selector2!: SelectornumericoComponent;

fijarSelector1(valor:number) {

this.selector1.fijar(valor);

}

fijarSelector2(valor:number) {

this.selector2.fijar(valor);

}

}

Debemos definir una propiedad llamada selector1 y mediante el decorador @ViewChild hacemos referencia a la variable definida en el template HTML que también se llama 'selector1':

@ViewChild('selector1') selector1!: SelectornumericoComponent;

Indicamos que sabemos que la propiedad no será nula con el operador "!" en la declaración de la propiedad. Esto se llama "assertion" o "no nulo" y le dice al compilador de TypeScript que confíe en nosotros.

Mediante las propiedades selector1 y selector2 tenemos las referencias de las dos componentes hijas, luego podemos llamar a sus métodos o acceder a sus propiedades:

fijarSelector1(valor:number) {

this.selector1.fijar(valor);

}

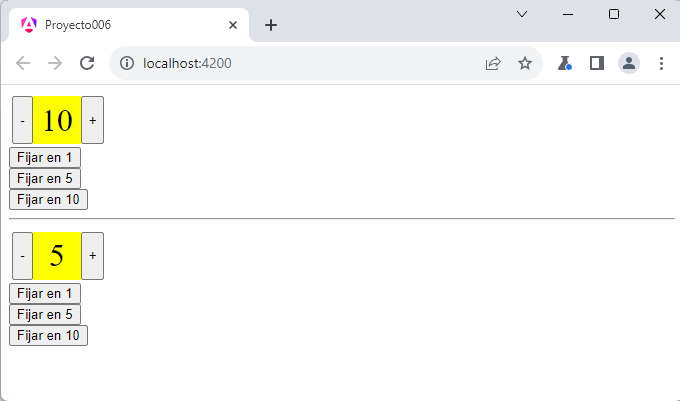
fijarSelector2(valor:number) {

this.selector2.fijar(valor);

}

Si ejecutamos ahora el proyecto:

ng server -o



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto006/).

1. Formularios: formulario basado en plantillas

Angular proporciona dos enfoques diferentes para manejar formularios:

1. Formularios basados en plantillas
2. Formularios reactivos

Ambos capturan los eventos de entrada del usuario desde la vista, crean un modelo de formulario y un modelo de datos para actualizar y proporcionan una forma de rastrear los cambios.

Los formularios basados en plantillas son útiles para agregar un formulario simple a una aplicación. Son más fáciles de agregar que los formularios reactivos, pero no escalan tan bien como los formularios reactivos. Si tiene requisitos de formulario y lógica muy básica podemos utilizar sin problemas formularios basados en plantillas.

Veremos en este concepto como implementar en Angular formularios basados en plantillas.

Formulario reactivo (directiva ngModel)

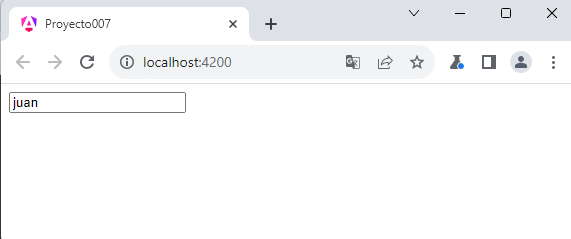
Por ejemplo, si en la clase AppComponent tenemos la propiedad 'nombre' con el valor 'juan':

nombre='juan';

Luego en la vista definimos la directiva ngModel entre corchetes y le asignamos el nombre de la propiedad definida en la clase:

<input type="text" [ngModel]="nombre">

Cuando arrancamos la aplicación podemos observar que el control input aparece automáticamente con el valor 'juan':



Para utilizar la directiva ngModel debemos importar el módulo 'FormsModule':

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

import { RouterOutlet } from '@angular/router';

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

@Component({

selector: 'app-root',

standalone: true,

imports: [RouterOutlet, **FormsModule**],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

nombre = 'juan';

}

Lo que hay que tener en cuenta es que es un enlace en una única dirección: el valor de la propiedad de la clase se refleja en la interfaz visual. Si el operador cambia el contenido del control 'input' por ejemplo por el nombre 'ana' luego la propiedad 'nombre' de la clase sigue almacenando el valor 'juan'.

Si queremos que el enlace sea en las dos direcciones debemos utilizar la siguiente sintaxis:

<input type="text" [(ngModel)]="nombre">

Un primer ejemplo muy corto que podemos hacer es crear un proyecto (proyecto007) para que se ingrese el nombre y apellido de una persona y se muestre inmediatamente en la parte inferior.

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

import { RouterOutlet } from '@angular/router';

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

@Component({

selector: 'app-root',

imports: [RouterOutlet, FormsModule],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

nombre = '';

apellido = '';

}

El archivo app.component.html donde definimos las directiva ngModel para cada control es:

<div>

<p>Ingrese nombre<input type="text" [(ngModel)]="nombre"></p>

<p>Ingrese apellido<input type="text" [(ngModel)]="apellido"></p>

<p>Nombre completo:{{nombre}},{{apellido}}</p>

</div>

<router-outlet />

Ejecutemos nuestra aplicación desde la línea de comandos de Node.js.

En el navegador nos muestra como se actualizan las propiedades cada vez que ingresamos un caracter en los controles 'input':



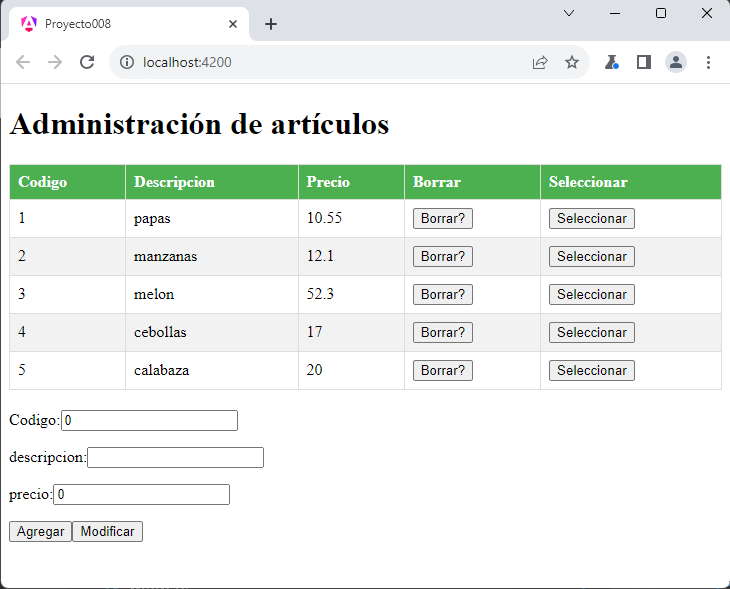
Es importante entender que hay un enlace en ambas direcciones. Si cambiamos el valor mediante el ingreso de datos por teclado en el control input, luego se ven reflejados en forma automática en las propiedades de la clase. De la misma forma si cambiamos el valor de la propiedad en la clase, luego se refleja en forma automática el valor en el formulario.

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto007/).

Problema

Confeccionar una aplicación que permita administrar un vector de objetos que almacena en cada elemento el código, descripción y precio de un artículo. Se debe poder agregar, borrar y modificar los datos de un artículo.

La interfaz visual de la aplicación debe ser similar a esta:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto008/).

Crearemos el proyecto008 para resolver este problema.

1. Lo primero que debemos hacer es desde la línea de comandos de Node.js proceder a crear el proyecto008:
2. ng new proyecto008
3. Para facilidad por el momento crearemos todo en una única componente, es decir en la que se crea automáticamente al crear el proyecto.

En el archivo app.component.css implementamos la hoja de estilo para la tabla HTML:

/\* Estilo para la tabla \*/

table {

width: 100%;

border-collapse: collapse;

margin-bottom: 20px;

}

/\* Estilo para las celdas de encabezado \*/

th, td {

border: 1px solid #dddddd;

padding: 8px;

text-align: left;

}

/\* Estilo alternado para filas \*/

tr:nth-child(even) {

background-color: #f2f2f2;

}

/\* Estilo para el encabezado de la tabla \*/

th {

background-color: #4CAF50;

color: white;

}

En el archivo app.component.html es donde mostramos una tabla HTML y un formulario para ingresar datos:

<div>

<h1>Administración de artículos</h1>

@if (hayRegistros()) {

<table>

<thead>

<tr>

<th>Codigo</th>

<th>Descripcion</th>

<th>Precio</th>

<th>Borrar</th>

<th>Seleccionar</th>

</tr>

</thead>

<tbody>

@for(art of articulos; track art.codigo) {

<tr>

<td>{{art.codigo}}</td>

<td>{{art.descripcion}}</td>

<td>{{art.precio}}</td>

<td><button (click)="borrar(art.codigo)">Borrar?</button></td>

<td><button (click)="seleccionar(art)">Seleccionar</button></td>

</tr>

}

</tbody>

</table>

} @else {

<p>No hay articulos.</p>

}

<div>

<p>

Codigo:<input type="number" [(ngModel)]="art.codigo" />

</p>

<p>

descripcion:<input type="text" [(ngModel)]="art.descripcion" />

</p>

<p>

precio:<input type="number" [(ngModel)]="art.precio" />

</p>

<p>

<button (click)="agregar()">Agregar</button>

<button (click)="modificar()">Modificar</button>

</p>

</div>

</div>

<router-outlet />

La clase app.component.ts tenemos:

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

import { RouterOutlet } from '@angular/router';

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

@Component({

selector: 'app-root',

imports: [RouterOutlet, FormsModule],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

art = {

codigo: 0,

descripcion: "",

precio: 0

}

articulos = [{ codigo: 1, descripcion: 'papas', precio: 10.55 },

{ codigo: 2, descripcion: 'manzanas', precio: 12.10 },

{ codigo: 3, descripcion: 'melon', precio: 52.30 },

{ codigo: 4, descripcion: 'cebollas', precio: 17 },

{ codigo: 5, descripcion: 'calabaza', precio: 20 },

];

hayRegistros() {

return this.articulos.length > 0;

}

borrar(codigo: number) {

for (let x = 0; x < this.articulos.length; x++)

if (this.articulos[x].codigo == codigo) {

this.articulos.splice(x, 1);

return;

}

}

agregar() {

if (this.art.codigo == 0) {

alert('Debe ingresar un código de articulo distinto a cero');

return;

}

for (let x = 0; x < this.articulos.length; x++)

if (this.articulos[x].codigo == this.art.codigo) {

alert('ya existe un articulo con dicho codigo');

return;

}

this.articulos.push({

codigo: this.art.codigo,

descripcion: this.art.descripcion,

precio: this.art.precio

});

this.art.codigo = 0;

this.art.descripcion = "";

this.art.precio = 0;

}

seleccionar(art: { codigo: number; descripcion: string; precio: number; }) {

this.art.codigo = art.codigo;

this.art.descripcion = art.descripcion;

this.art.precio = art.precio;

}

modificar() {

for (let x = 0; x < this.articulos.length; x++)

if (this.articulos[x].codigo == this.art.codigo) {

this.articulos[x].descripcion = this.art.descripcion;

this.articulos[x].precio = this.art.precio;

return;

}

alert('No existe el código de articulo ingresado');

}

}

Ejecutemos la aplicación desde la ventana de Node.js mediante el comando:

ng serve -o

Listado

Pasemos a analizar las distintas partes de nuestra aplicación. Los archivos app.component.ts y app.component.html están totalmente integrados y con objetivos bien definidos cada uno. El archivo '.html' almacena la vista y el archivo '.ts' almacena el modelo de datos.

Definimos en el modelo (app.component.ts) un vector de objetos llamado 'articulos' y almacenamos 5 elementos:

articulos = [{ codigo: 1, descripcion: 'papas', precio: 10.55 },

{ codigo: 2, descripcion: 'manzanas', precio: 12.10 },

{ codigo: 3, descripcion: 'melon', precio: 52.30 },

{ codigo: 4, descripcion: 'cebollas', precio: 17 },

{ codigo: 5, descripcion: 'calabaza', precio: 20 },

];

En la vista (app.component.html) verificamos con un if si el vector tienen elementos:

@if (hayRegistros()) {

Como vemos llamamos al método 'hayRegistros()' que se encuentra implementado en el archivo \*.ts:

hayRegistros() {

return this.articulos.length > 0;

}

Si retorna true luego generamos una tabla HTML que muestre los datos del modelo y lo recorremos mediante un @for:

<table>

<thead>

<tr>

<th>Codigo</th>

<th>Descripcion</th>

<th>Precio</th>

<th>Borrar</th>

<th>Seleccionar</th>

</tr>

</thead>

<tbody>

@for(art of articulos; track art.codigo) {

<tr>

<td>{{art.codigo}}</td>

<td>{{art.descripcion}}</td>

<td>{{art.precio}}</td>

<td><button (click)="borrar(art.codigo)">Borrar?</button></td>

<td><button (click)="seleccionar(art)">Seleccionar</button></td>

</tr>

}

</tbody>

</table>

Disponemos en cada fila dos botones y definimos sus respectivos eventos 'click' para que al ser presionados llamen a métodos del modelo para borrar o seleccionar el artículo respectivo. Los métodos envían como parámetro el artículo para saber cual borrar o seleccionar.

En la vista disponemos una serie de 'input' que nos permiten ingresar el código, descripción y precio de un artículo:

<div>

<p>

Codigo:<input type="number" [(ngModel)]="art.codigo" />

</p>

<p>

descripcion:<input type="text" [(ngModel)]="art.descripcion" />

</p>

<p>

precio:<input type="number" [(ngModel)]="art.precio" />

</p>

<p>

<button (click)="agregar()">Agregar</button>

<button (click)="modificar()">Modificar</button>

</p>

</div>

Agregado

Podemos comprobar que los controles HTML tienen la directiva 'ngModel' bidireccional, es decir que cuando el operador carga un dato en el primer 'input' se actualiza automáticamente en el modelo el dato cargado en 'art.codigo':

Podemos comprobar que en el modelo tenemos definido un objeto llamado art con tres propiedades:

art={

codigo:0 ,

descripcion:"",

precio:0

}

Al presionar el botón agregar se ejecuta el método 'agregar':

agregar() {

if (this.art.codigo == 0) {

alert('Debe ingresar un código de articulo distinto a cero');

return;

}

for (let x = 0; x < this.articulos.length; x++)

if (this.articulos[x].codigo == this.art.codigo) {

alert('ya existe un articulo con dicho codigo');

return;

}

this.articulos.push({

codigo: this.art.codigo,

descripcion: this.art.descripcion,

precio: this.art.precio

});

this.art.codigo = 0;

this.art.descripcion = "";

this.art.precio = 0;

}

En este método primero recorremos el vector articulos para comprobar si hay algún otro artículo con el mismo código. En el caso que no exista procedemos a añadir un nuevo elemento llamando al método push y pasando un objeto que creamos en dicho momento con los datos almacenados en el objeto 'art' que se encuentra enlazado con el formulario.

Luego asignamos cero y cadena vacía a todas las propiedades del objeto art con el objetivo de borrar todos los 'input' del formulario.

Al agregar un elemento al vector 'Angular' se encarga de actualizar la vista sin tener que indicar nada en nuestro código.

Borrado

Cuando se presiona el botón de borrar se ejecuta el método 'borrar':

borrar(codigo: number) {

for (let x = 0; x < this.articulos.length; x++)

if (this.articulos[x].codigo == codigo) {

this.articulos.splice(x, 1);

return;

}

}

Recorremos el vector y controlamos uno a uno el código del artículo seleccionado con cada uno de los elementos del vector. El que coincide lo eliminamos del vector llamando al método splice indicando la posición y cuantas componentes borrar a partir de ese.

Selección

Cuando se presiona el botón de seleccionar se ejecuta el método 'seleccionar':

seleccionar(art: { codigo: number; descripcion: string; precio: number; }) {

this.art.codigo=art.codigo;

this.art.descripcion=art.descripcion;

this.art.precio=art.precio;

}

Lo único que hacemos es actualizar el objeto art del modelo con el artículo que acaba de seleccionar el operador (llega como parámetro el artículo seleccionado)

Modificación

Cuando presiona el botón de modificación se ejecuta el método:

modificar() {

for(let x=0;x<this.articulos.length;x++)

if (this.articulos[x].codigo==this.art.codigo)

{

this.articulos[x].descripcion=this.art.descripcion;

this.articulos[x].precio=this.art.precio;

return;

}

alert('No existe el código de articulo ingresado');

}

Buscamos el código de articulo del control 'input' dentro del vector, en caso de encontrarlo procedemos a modificar la descripción y precio.

Tener en cuenta que por ahora estamos haciendo aplicaciones que almacenan sus datos en forma local y no estamos enviando los mismos a un servidor, luego veremos distintas alternativas de enviar y recuperar datos de un servidor.

Recordar que podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto008/).

1. Formularios reactivos : ReactiveFormsModule, FormControl

Hemos dicho que Angular proporciona dos enfoques diferentes para manejar formularios:

1. Basado en plantillas (visto en el concepto anterior)
2. Reactivos

Ambos capturan los eventos de entrada del usuario desde la vista, crean un modelo de formulario y un modelo de datos para actualizar y proporcionan una forma de rastrear los cambios. Hasta ahora nosotros hemos trabajado con el modelo basado en plantillas.

El enfoque reactivo es más robusto, son más escalables, reutilizables y comprobables. Si los formularios son una parte clave de su aplicación, conviene utilizar el modelo reactivo.

Diferencias entre ambos modelos

* La configuración con el modelo de plantilla se hace mediante directivas y es menos explícito que el modelo reactivo que debe configurarse en la clase mediante código.
* En el formulario basado en plantillas el modelo de datos es más desestructurado.
* En el formulario basado en plantilla la validación se hace mediante directivas, en cambio en el modelo reactivo se hace mediante funciones.
* Las pruebas unitarias son más faciles de implementar en los formularios reactivos.
* Maneja cualquier escenario complejo es más fácil implementarlo con formularios reactivos.

Para trabajar con formularios reactivos necesitamos importar la clase 'ReactiveFormsModule' y crear un objeto de la clase FormControl por cada control visual que contendrá nuestro formulario.

Nuestro primer problema con formularios reactivos será el más elemental, ya que contendrá un solo control 'input' para el ingreso de una cadena.

Problema

Confeccionar una aplicación que permita ingresar actividades pendientes mediante un control 'input' de tipo 'text'.

Se deben listar todas las actividades ingresadas hasta el momento, además de poder borrar una en particular o todas a la vez.

Almacenar en forma local en el navegador las actividades mediante el API localStorage para evitar que se pierdan las actividades cuando se cierre el navegador.

Crearemos primero el proyecto:

ng new proyecto009

Trabajaremos con la componente que generar automáticamente Angular CLI cuando creamos el proyecto, pasamos a modificar la vista (app.component.html):

<p> Actividad:

<input type="text" [formControl]="actividad">

</p>

<p><button (click)="agregar()">Agregar</button></p>

<ol>

@for(acti of lista;track $index) {

<li>

{{acti}} <a (click)="borrar($index)" href="#">Borra?</a>

</li>

}

</ol>

<p><button (click)="borrarTodas()">Borrar todas las actividades</button></p>

<router-outlet />

Analizaremos este archivo en conjunto luego de presentar 'app.component.ts'

La clase asociada a la vista 'app.component.html' es el archivo 'app.component.ts' donde implementamos la siguiente lógica:

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, FormControl } from '@angular/forms';

@Component({

selector: 'app-root',

imports: [ RouterOutlet, ReactiveFormsModule,],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

actividad = new FormControl();

lista: string[];

constructor() {

this.lista = [];

let datos = localStorage.getItem("actividades");

if (datos != null) {

let arreglo = JSON.parse(datos);

if (arreglo != null)

for (let actividad of arreglo)

this.lista.push(actividad);

}

}

agregar() {

this.lista.push(this.actividad.value);

localStorage.setItem('actividades', JSON.stringify(this.lista));

this.actividad.setValue('');

}

borrar(pos: number) {

this.lista.splice(pos, 1);

localStorage.clear();

localStorage.setItem('actividades', JSON.stringify(this.lista));

}

borrarTodas() {

localStorage.clear();

this.lista = [];

}

}

/\*Archivo corregido por ChatGPT, porque el original da error::

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, FormControl } from '@angular/forms';

@Component({

selector: 'app-root',

standalone: true,

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

actividad = new FormControl();

lista: string[] = [];

constructor() {

// Tres actividades iniciales hardcodeadas

this.lista = [

'Leer un libro',

'Hacer ejercicio',

'Aprender Angular'

];

}

agregar() {

const valor = this.actividad.value;

if (typeof valor === 'string' && valor.trim() !== '') {

this.lista.push(valor.trim());

this.actividad.setValue('');

}

}

borrar(pos: number) {

this.lista.splice(pos, 1);

}

borrarTodas() {

this.lista = [];

}

}

Lo primero que hacemos es importar la clase 'ReactiveFormsModule' y la interface 'FormControl':

import { ReactiveFormsModule, FormControl } from '@angular/forms';

Debemos importar en la componente la clase 'ReactiveFormsModule':

@Component({

selector: 'app-root',

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

Dentro de la clase 'AppComponent' creamos un objeto de tipo 'FormControl' llamado 'actividad' (uno solo ya que solo tenemos un control de entrada de datos):

actividad = new FormControl();

Este atributo 'actividad' se asocia con la vista mediante la sintaxis:

<input type="text" [formControl]="actividad">

La cadena que carga el operador en el control visual luego se la puede acceder mediante la propiedad 'value' y si queremos cambiar su contenido disponemos del método 'setValue'.

El problema requiere que se almacenen todas las actividades que ingresa el usuario por teclado, esto sucede en el arreglo 'lista'. La definimos dentro de la clase:

lista: string[];

Cuando el operador presiona el botón de 'Agregar' se llama el método 'agregar' y procedemos a recuperar el dato del control visual mediante la propiedad value y la agregamos al arreglo:

this.lista.push(this.actividad.value);

Inmediatamente procedemos a borrar el contenido del control visual llamando al método 'setValue':

this.actividad.setValue('');

Nuestro problema requiere almacenar cada una de las actividades en un arreglo que se pasa a mostrar en la vista mediante la estructura @for:

<ol>

@for(acti of lista;track $index) {

<li>

{{acti}} <a (click)="borrar($index)" href="#">Borra?</a>

</li>

}

</ol>

Debemos especificar la variable especial $index que se numera de 0 en adelante. Dicho valor se pasa al método borrar: (click)="borrar($index)"

* Vimos hasta ahora todo lo relacionado con el control FormControl, pero nuestro problema requiere que la lista de actividades no se borre al cerrar el navegador, para ello utilizamos el API localStorage que proporciona HTML5.

Cuando se presiona el botón 'agregar', luego de guardar la actividad del formulario procedemos a almacenar mediante el método 'setItem' del objeto 'localStorage' los datos de la lista pero en formato JSON:

agregar() {

this.lista.push(this.actividad.value);

localStorage.setItem('actividades', JSON.stringify(this.lista));

this.actividad.setValue('');

}

Si se presiona el botón de borrar todas las actividades procedemos por un lado a borrar los elementos del arreglo y además a borrar los datos almacenados en 'localStorage':

borrarTodas() {

localStorage.clear();

this.lista=[];

}

Otro punto importante es que cuando se ejecuta el método constructor al cargar la componente, verificamos si hay datos almacenados en el 'localStorage' para leerlos y cargar el arreglo 'lista' con los datos previos de ejecuciones anteriores de la aplicación Angular:

constructor() {

this.lista = [];

let datos = localStorage.getItem("actividades");

if (datos != null) {

let arreglo = JSON.parse(datos);

if (arreglo != null)

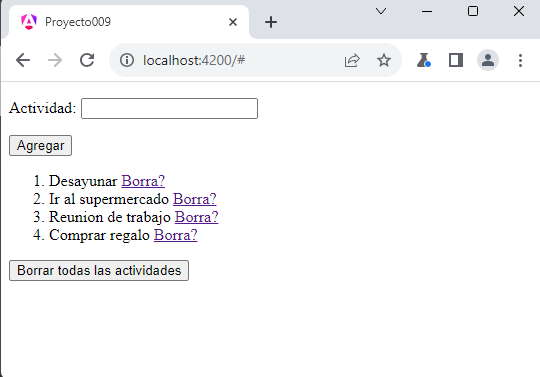
for (let actividad of arreglo)

this.lista.push(actividad);

}

}

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto009/).

1. Formularios reactivos: FormGroup

Si tenemos un solo control de entrada de datos podemos como en el concepto anterior definir únicamente un objeto de la clase FormControl, pero un formulario que contiene un conjunto de controles la manera más efectiva es agruparlos definiendo un objeto de tipo FormGroup.

Problema

Confeccionar un formulario de contacto que permita cargar el nombre, mail y un mensaje. Cuando se presiona un botón mostrar mediante interpolación los datos cargados.

Crearemos primero el proyecto:

ng new proyecto010

Modificamos la vista de la componente que muestra el formulario reactivo (app.component.html):

<form [formGroup]="formularioContacto" (ngSubmit)="submit()">

<p>Nombre:

<input type="text" formControlName="nombre">

</p>

<p>Mail:

<input type="text" formControlName="mail">

</p>

<p>Mensaje<br>

<textarea rows="10" cols="70" formControlName="mensaje"></textarea>

</p>

<button type="submit">Confirmar</button>

</form>

<div>{{datos}}</div>

<router-outlet />

Analizaremos este archivo en conjunto luego de presentar 'app.component.ts'

La clase asociada a la vista es (app.component.ts):

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, FormControl, FormGroup } from '@angular/forms';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

datos='';

formularioContacto = new FormGroup({

nombre: new FormControl(''),

mail: new FormControl(''),

mensaje: new FormControl('')

});

submit() {

this.datos = `Nombre=${this.formularioContacto.value.nombre}

Mail=${this.formularioContacto.value.mail}

Mensaje=${this.formularioContacto.value.mensaje}

`;

}

}

Importamos ahora:

import { ReactiveFormsModule, FormControl, FormGroup } from '@angular/forms';

Creamos un objeto de la clase FormGroup y le pasamos al constructor del mismo un objeto literal con la creación de un objeto de la clase FormControl por cada control visual de la vista:

formularioContacto = new FormGroup({

nombre: new FormControl(''),

mail: new FormControl(''),

mensaje: new FormControl('')

});

Podemos pasar a cada constructor el valor inicial que tenga el control visual, en nuestro caso hemos pasado string vacíos.

En la vista agrupamos todos los controles mediante la etiqueta 'form' y definimos la directiva [formGroup] asignando la variable de tipo FormGroup definida en la clase:

<form [formGroup]="formularioContacto" (ngSubmit)="submit()">

Asociamos el evento (ngSubmit) con el método de la clase que captura el clic del botón del formulario.

Cada uno de los controles requiere definir la propiedad 'formControlName' con los nombres asignados en la clase cuando creamos cada FormControl:

<p>Nombre:

<input type="text" formControlName="nombre">

</p>

<p>Mail:

<input type="text" formControlName="mail">

</p>

<p>Mensaje<br>

<textarea rows="10" cols="70" formControlName="mensaje"></textarea>

</p>

El método 'submit' procede a recuperar cada valor ingresado en el formulario y lo almacena en el atributo 'datos':

submit() {

this.datos=`Nombre=${this.formularioContacto.value.nombre}

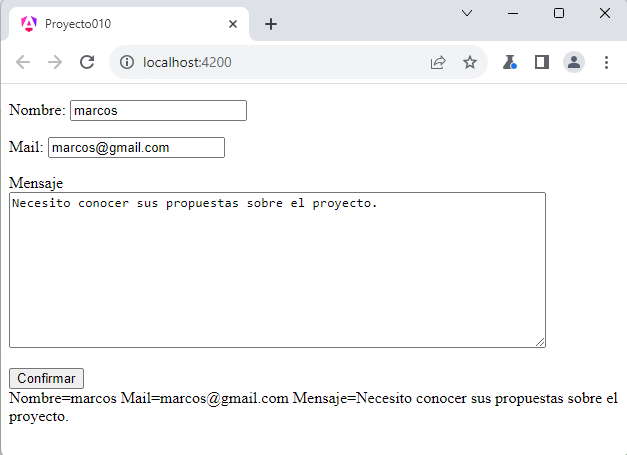
Mail=${this.formularioContacto.value.mail}

Mensaje=${this.formularioContacto.value.mensaje}`;

El atributo datos se muestra en la vista mediante interpolación:

<div>{{datos}}</div>

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto010/).

1. Formularios reactivos : controles checkbox, radio y select

Vimos en los conceptos anteriores como trabajar con formularios reactivos en Angular, pero siempre utilizando un input de tipo text o un textarea. Ahora veremos como configurar los controles checkbox, radio y select.

Problema

Confeccionar un formulario que permita al usuario ingresar un valor decimal y luego pueda seleccionar:

Mediante dos controles radio puede indicar si quiere la conversión a hexadecimal u octal.

Mediante un control select indica si quiere calcular el valor ingresado elevado a la potencia 2, 3 o 4.

Finalmente con un control checkbox puede indicar que muestre la cantidad de dígitos del valor ingresado.

Crearemos primero el proyecto:

ng new proyecto011

Modificamos la vista de la componente que muestra el formulario reactivo (app.component.html):

<form [formGroup]="formularioConversion" (ngSubmit)="submit()">

<p>Ingrese número decimal:

<input type="text" formControlName="numerodecimal">

</p>

<p>Convertir a:<br>

<input type="radio" formControlName="base" value="octal">Octal

<input type="radio" formControlName="base" value="hexadecimal">Hexadecimal

</p>

<p>Elevar a:<br>

<select formControlName="potencia">

<option>2</option>

<option>3</option>

<option>4</option>

</select>

</p>

<p><input type="checkbox" formControlName="largo">Mostrar cantidad de dígitos

</p>

<p><button type="submit">Confirmar</button></p>

</form>

<div>

<p>Conversion:{{conversion}}</p>

<p>Potencia:{{potenciaNumero}}</p>

<p>Cantidad de dígitos:{{cantidad}}</p>

</div>

<router-outlet />

Analizaremos este archivo en conjunto luego de presentar 'app.component.ts'

La clase asociada a la vista es (app.component.ts):

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, FormControl, FormGroup } from '@angular/forms';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

conversion = '';

potenciaNumero = '';

cantidad = '';

formularioConversion = new FormGroup({

numerodecimal: new FormControl(''),

base: new FormControl('octal'),

potencia: new FormControl('2'),

largo: new FormControl(true)

});

submit() {

if (this.formularioConversion.value.numerodecimal) {

if (this.formularioConversion.value.base == "hexadecimal")

this.conversion = parseInt(this.formularioConversion.value.numerodecimal).toString(16);

if (this.formularioConversion.value.base == "octal")

this.conversion = parseInt(this.formularioConversion.value.numerodecimal).toString(8);

if (this.formularioConversion.value.numerodecimal && this.formularioConversion.value.potencia)

this.potenciaNumero = Math.pow(parseInt(this.formularioConversion.value.numerodecimal), parseInt(this.formularioConversion.value.potencia)).toString();

if (this.formularioConversion.value.largo)

this.cantidad = this.formularioConversion.value.numerodecimal.length.toString();

}

}

}

Creamos un objeto de la clase FormGroup y le pasamos un objeto literal con la referencia a los 4 controles de formulario que necesitamos:

formularioConversion = new FormGroup({

numerodecimal: new FormControl(''),

base: new FormControl('octal'),

potencia: new FormControl('2'),

largo: new FormControl(true)

});

El atributo base se asocia a los controles 'radio' de la vista, he indicamos que aparezca seleccionado por defecto aquel que almacena en value el valor 'octal':

<p>Convertir a:<br>

<input type="radio" formControlName="base" value="octal">Octal

<input type="radio" formControlName="base" value="hexadecimal">Hexadecimal

</p>

El control HTML select se asocia al atributo 'potencia', indicando que se muestre seleccionada la opción con valor '2':

potencia: new FormControl('2'),

La vista para este FormControl es:

<p>Elevar a:<br>

<select formControlName="potencia">

<option>2</option>

<option>3</option>

<option>4</option>

</select>

</p>

Finalmente para el control checkbox indicamos que se muestre seleccionado por defecto al cargar el formulario:

largo: new FormControl(true)

Cuando se presiona el botón se ejecuta el método 'submit', donde efectuamos las distintas operaciones según los valores seleccionados en el formulario:

submit() {

if (this.formularioConversion.value.numerodecimal) {

if (this.formularioConversion.value.base == "hexadecimal")

this.conversion = parseInt(this.formularioConversion.value.numerodecimal).toString(16);

if (this.formularioConversion.value.base == "octal")

this.conversion = parseInt(this.formularioConversion.value.numerodecimal).toString(8);

if (this.formularioConversion.value.numerodecimal && this.formularioConversion.value.potencia)

this.potenciaNumero = Math.pow(parseInt(this.formularioConversion.value.numerodecimal), parseInt(this.formularioConversion.value.potencia)).toString();

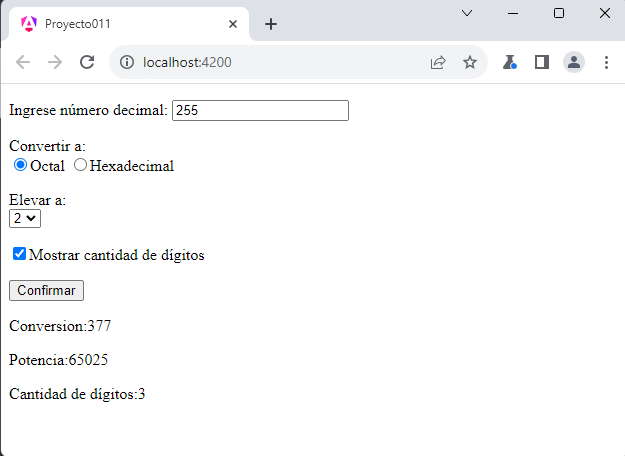
if (this.formularioConversion.value.largo)

this.cantidad = this.formularioConversion.value.numerodecimal.length.toString();

}

}

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto011/).

1. Formularios reactivos : FormGroup anidados

Cuando trabajamos con formularios reactivos en Angular normalmente creamos un FormGroup para agrupar los controles visuales que se les asocian un objeto de la clase FormControl a cada uno.

Podemos crear dentro de un FormGroup otro FormGroup que agrupe un conjunto de controles relacionados.

Problema

Confeccionar un formulario que permita cargar los siguientes datos de un alumno: dni, nombre, materia y sus tres notas. Agrupar las tres notas dentro de un FormGroup anidado al principal.

Al presionar un botón mostrar un mensaje si se encuentra aprobado (todas las notas mayores o iguales a 4)

Crearemos primero el proyecto:

ng new proyecto012

Modificamos la vista de la componente que muestra el formulario reactivo (app.component.html):

<form [formGroup]="formAlumno" (ngSubmit)="submit()">

<p>Dni:

<input type="text" formControlName="dni">

</p>

<p>Nombre:

<input type="text" formControlName="nombre">

</p>

<div formGroupName="notas">

<p>Primer nota:

<input type="text" formControlName="nota1">

</p>

<p>Segunda nota:

<input type="text" formControlName="nota2">

</p>

<p>Tercer nota:

<input type="text" formControlName="nota3">

</p>

</div>

<p><button type="submit">Confirmar</button></p>

</form>

<div>

<p>Resultado:{{resultado}}</p>

</div>

<router-outlet />

Analizaremos este archivo en conjunto luego de presentar 'app.component.ts'

La clase asociada a la vista es (app.component.ts):

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, FormControl, FormGroup } from '@angular/forms';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

resultado = '';

formAlumno = new FormGroup({

dni: new FormControl(''),

nombre: new FormControl(''),

notas: new FormGroup({

nota1: new FormControl(''),

nota2: new FormControl(''),

nota3: new FormControl('')

})

});

submit() {

if (this.formAlumno.value.notas) {

if (this.formAlumno.value.notas.nota1 &&

this.formAlumno.value.notas.nota2 &&

this.formAlumno.value.notas.nota3) {

let nota1 = parseInt(this.formAlumno.value.notas.nota1);

let nota2 = parseInt(this.formAlumno.value.notas.nota2);

let nota3 = parseInt(this.formAlumno.value.notas.nota3);

if (nota1 >= 4 && nota2 >= 4 && nota3 >= 4)

this.resultado = "El alumno queda aprobado por esas notas";

else

this.resultado = "El alumno no aprueba por esas notas";

}

}

}

}

Creamos un objeto de la clase FormGroup y dentro de este otro objeto de la clase FormGroup que agrupa las tres notas:

formAlumno = new FormGroup({

dni: new FormControl(''),

nombre: new FormControl(''),

notas: new FormGroup({

nota1: new FormControl(''),

nota2: new FormControl(''),

nota3: new FormControl('')

})

});

Esto provoca en la vista que cuando asociemos las tres notas debemos agruparlas dentro de un etiqueta HTML que defina una propiedad 'formGroupName':

<div formGroupName="notas">

<p>Primer nota:

<input type="text" formControlName="nota1">

</p>

<p>Segunda nota:

<input type="text" formControlName="nota2">

</p>

<p>Tercer nota:

<input type="text" formControlName="nota3">

</p>

</div>

Cuando se presiona el botón se verifica si las tres notas ingresadas son mayores o iguales a 4 para mostrar por pantalla si el alumno se encuentra aprobado o no (mediante los if verificamos que se hayan cargado valores en los controles input):

submit() {

if (this.formAlumno.value.notas) {

if (this.formAlumno.value.notas.nota1 &&

this.formAlumno.value.notas.nota2 &&

this.formAlumno.value.notas.nota3) {

let nota1 = parseInt(this.formAlumno.value.notas.nota1);

let nota2 = parseInt(this.formAlumno.value.notas.nota2);

let nota3 = parseInt(this.formAlumno.value.notas.nota3);

if (nota1 >= 4 && nota2 >= 4 && nota3 >= 4)

this.resultado = "El alumno queda aprobado por esas notas";

else

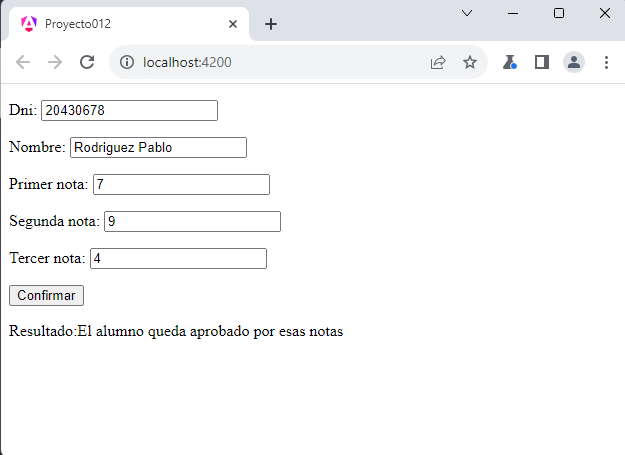
this.resultado = "El alumno no aprueba por esas notas";

}

}

}

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto012/).

1. Formularios reactivos : validaciones estándares de Angular

Una funcionalidad muy importante de un formulario es la validación de las entradas de datos del usuario y la información que se le provee al mismo indicando los datos mal cargados.

Cuando utilizamos formularios reactivos la lógica de la validación se encuentra en la clase (archivo \*.js) y no en la vista (archivo \*.html)

El framework de Angular provee una clase llamada 'Validators' que contiene una serie de métodos estáticos para validar entradas de datos muy comunes como la estructura de un email, valores numéricos máximos y mínimos, cantidad mínima y máxima de caracteres etc.

La asignación de los métodos de validación se hace cuando se crea el objeto de la clase FormControl. Veremos con un ejemplo como configuramos distintos métodos de validación a los controles de un formulario.

Problema

Confeccionar un formulario de contacto que permita cargar el nombre, mail y un mensaje. Implementar las siguientes validaciones a los controles del formulario:

* Los tres controles no pueden quedar vacíos.
* El nombre debe tener como mínimo 10 caracteres.
* El email ingresado debe ser correcto.
* El mensaje no puede superar los 500 caracteres.

Cuando se presione un botón mostrar un mensaje indicando si todos los controles se encuentran correctamente cargados.

Crearemos primero el proyecto:

ng new proyecto013

Modificamos la vista de la componente que muestra el formulario reactivo y eventualmente los mensajes de error de entrada de datos (app.component.html):

<form [formGroup]="formularioContacto" (ngSubmit)="submit()">

<p>Nombre:

<input type="text" formControlName="nombre" required>

@if(this.formularioContacto.get('nombre')?.errors?.['required']) {

<span>(El nombre no puede quedar vacío)</span>

}

@if(this.formularioContacto.get('nombre')?.errors?.['minlength']) {

<span>(Debe tener como mínimo 10 caracteres)</span>

}

</p>

<p>Mail:

<input type="text" formControlName="mail">

@if(this.formularioContacto.get('mail')?.errors?.['required']) {

<span>(El mail no puede quedar vacío)</span>

}

@if(this.formularioContacto.get('mail')?.errors?.['email']) {

<span>(El mail no es válido)</span>

}

</p>

<p>Mensaje<br>

<textarea rows="10" cols="70" formControlName="mensaje"></textarea>

</p>

@if(this.formularioContacto.get('mensaje')?.errors?.['required']) {

<div>(El mensaje no puede quedar vacío)</div>

}

@if(this.formularioContacto.get('mensaje')?.errors?.['maxlength']) {

<div>(Debe tener como máximo 500 caracteres)</div>

}

<button type="submit">Confirmar</button>

</form>

<div>{{resultado}}</div>

<router-outlet />

Analizaremos este archivo en conjunto luego de presentar 'app.component.ts'

La clase asociada a la vista es (app.component.ts):

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, FormControl, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

resultado='';

formularioContacto = new FormGroup({

nombre: new FormControl('', [Validators.required, Validators.minLength(10)]),

mail: new FormControl('', [Validators.required, Validators.email]),

mensaje: new FormControl('', [Validators.required, Validators.maxLength(500)])

});

submit() {

if (this.formularioContacto.valid)

this.resultado = "Todos los datos son válidos";

else

this.resultado = "Hay datos inválidos en el formulario";

}

}

Lo primero que debemos hacer es importar la clase 'Validators' y las otras que ya conocemos:

import { ReactiveFormsModule, FormControl, FormGroup, Validators } from '@angular/forms';

Cuando creamos los objetos de la clase FormControl debemos pasar como segundo parámetro del constructor los nombres de métodos que se le aplicarán a cada control de formulario:

formularioContacto = new FormGroup({

nombre: new FormControl('', [Validators.required, Validators.minLength(10)]),

mail: new FormControl('', [Validators.required, Validators.email]),

mensaje: new FormControl('', [Validators.required, Validators.maxLength(500)])

});

Por ejemplo para el 'nombre' configuramos que se apliquen las validaciones: [Validators.required, Validators.minLength(10)], luego en la vista podemos verificar que validaciones se cumplen y cuales no:

<p>Nombre:

<input type="text" formControlName="nombre" required>

@if(this.formularioContacto.get('nombre')?.errors?.['required']) {

<span>(El nombre no puede quedar vacío)</span>

}

@if(this.formularioContacto.get('nombre')?.errors?.['minlength']) {

<span>(Debe tener como mínimo 10 caracteres)</span>

}

</p>

Mediante comandos @if validamos si se debe mostrar el mensaje de error dependiendo si el atributo this.formularioContacto.get('nombre')?.errors? existe o no.

De forma similar se hacen las validaciones para los otros controles.

Cuando se presiona el botón analizamos la propiedad 'valid' del objeto FormGroup, si almacena true significa que todos los controles de validación del formulario se verifican correctos:

submit() {

if (this.formularioContacto.valid)

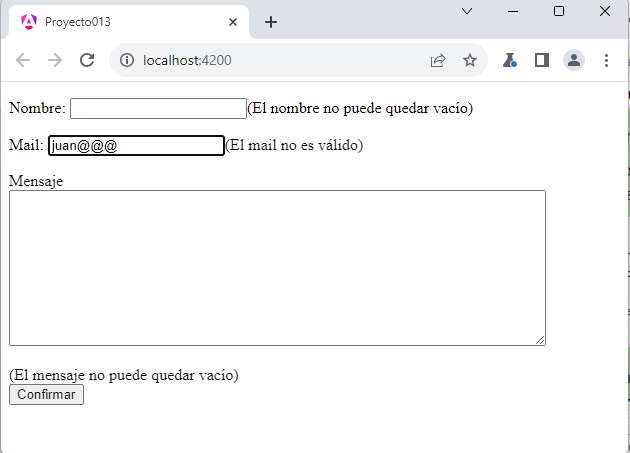
this.resultado = "Todos los datos son válidos";

else

this.resultado = "Hay datos inválidos en el formulario";

}

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto013/).

Acotaciones

Para conocer todos los métodos de validación que nos provee Angular podemos visitar su documentación oficial y analizar la clase [Validators](https://angular.dev/guide/forms/form-validation).

1. Formularios reactivos : validaciones personalizadas

Hemos visto que Angular provee una serie de funcionalidades para validar si un email es correcto, cantidad de caracteres máximos o mínimos ingresados, patrones de cadenas ingresados etc.  
Hay situaciones particulares en nuestros proyectos que no se adaptan a ninguna de esas funcionalidades, para esas situaciones podemos codificar nuestras propias funciones de validación.

Veremos los pasos a dar para crear nuestra propia función de validación.

Problema

Confeccionar un formulario que solicite el ingreso de un número múltiplo de 5. Validar que el valor ingresado sea múltiplo de 5 para que se active el botón de envío.

Crearemos primero el proyecto:

ng new proyecto014

La función de validación se la podría definir en el mismo archivo de la componente, pero si la misma se empleará en más de un formulario, lo más conveniente es separarla en una clase independiente.

Crearemos la clase 'ValidacionesPropias':

ng generate class ValidacionesPropias

Se genera el archivo 'validaciones-propias.ts' donde debemos implementar el algoritmo de validación y también se genera el archivo 'validaciones-propias.spec.ts' para implementar pruebas unitarias.

Codificaremos ahora el método 'multiplo5' en la clase 'ValidacionesPropias':

import { AbstractControl, ValidationErrors } from '@angular/forms';

export class ValidacionesPropias {

static multiplo5(control: AbstractControl): ValidationErrors| null {

let nro = parseInt(control.value);

if (nro % 5 == 0)

return null;

else

return { multiplo5: true }

}

}

El método debe retornar null si el dato es correcto, en nuestro caso si el valor ingresado en el 'FormControl' el resto de dividirlo por 5 genera un cero luego retornamos null, en caso contrario retornamos un tipo de dato ValidationErrors, definiendo un atributo con el nombre del método y como valor true.

Modificamos la vista de la componente que muestra el formulario reactivo y eventualmente el mensaje de error de entrada de datos (app.component.html):

<form [formGroup]="formularioContacto" (ngSubmit)="submit()">

<p>Ingrese un múltiplo de 5:

<input type="number" formControlName="numero">

@if(this.formularioContacto.get('numero')?.errors?.['multiplo5']) {

<span>(El valor debe ser multiplo de 5)</span>

}

</p>

<button type="submit" [disabled]="formularioContacto.invalid">Confirmar</button>

</form>

<router-outlet />

La clase asociada a la vista es (app.component.ts):

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, FormControl, FormGroup} from '@angular/forms';

import { ValidacionesPropias } from './validaciones-propias';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

formularioContacto = new FormGroup({

numero: new FormControl('', [ValidacionesPropias.multiplo5])

});

submit() {

alert('dato correcto');

}

}

Importamos la clase que acabamos de crear:

import { ValidacionesPropias } from './validaciones-propias';

Pasamos al constructor de la clase FormControl la referencia del método que se encargará de validar el dato ingresado, de forma similar a las validaciones estándares:

formularioContacto = new FormGroup({

numero: new FormControl('', [ValidacionesPropias.multiplo5])

});

En la vista mostramos el mensaje de error si existe el atributo, recordemos que si nuestra función de validación retorna null luego no existe en 'errors' el valor 'multiplo5':

@if(this.formularioContacto.get('numero')?.errors?.['multiplo5']) {

<span>(El valor debe ser multiplo de 5)</span>

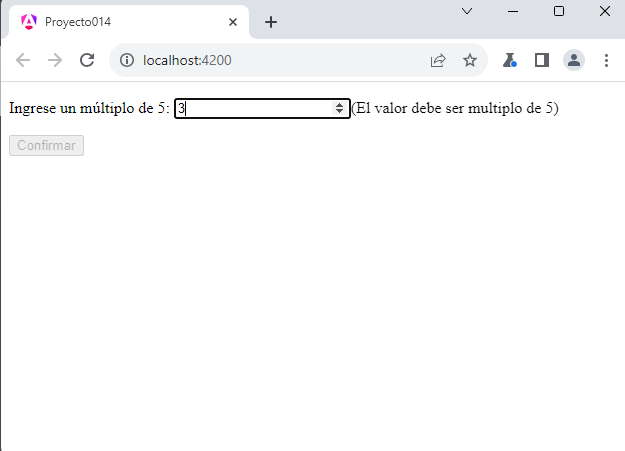
}

Para que el botón del formulario permanezca desactivado mientras alguno de los datos del formulario no pase el test de validación disponemos:

<button type="submit" [disabled]="formularioContacto.invalid">Confirmar</button>

'invalid' almacena true si alguno de los controles que estamos validando no es válido.

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto014/).

1. Formularios reactivos : FormBuilder

La clase FormBuilder nos permite crear objetos de la clase FormGroup y FormControl con una sintaxis más corta. Es decir que el resultado final será el mismo al que obtenemos con la sintaxis vista anteriormente.  
Queda en decisión del desarrollador utilizar una u otra sintaxis.

Resolveremos un ejercicio de conceptos anteriores pero ahora con la sintaxis propuesta con la clase FormBuilder.

Problema

Confeccionar un formulario de contacto que permita cargar el nombre, mail y un mensaje. Implementar las siguientes validaciones a los controles del formulario:

* Los tres controles no pueden quedar vacíos.
* El nombre debe tener como mínimo 10 caracteres.
* El email ingresado debe ser correcto.
* El mensaje no puede superar los 500 caracteres.

Cuando se presione un botón mostrar un mensaje indicando si todos los controles se encuentran correctamente cargados.

Crearemos primero el proyecto:

ng new proyecto015

Modificamos la vista de la componente que muestra el formulario reactivo y eventualmente los mensajes de error de entrada de datos (app.component.html):

<form [formGroup]="formularioContacto" (ngSubmit)="submit()">

<p>Nombre:

<input type="text" formControlName="nombre" required>

@if(this.formularioContacto.get('nombre')?.errors?.['required']) {

<span>(El nombre no puede quedar vacío)</span>

}

@if(this.formularioContacto.get('nombre')?.errors?.['minlength']) {

<span>(Debe tener como mínimo 10 caracteres)</span>

}

</p>

<p>Mail:

<input type="text" formControlName="mail">

@if(this.formularioContacto.get('mail')?.errors?.['required']) {

<span>(El mail no puede quedar vacío)</span>

}

@if(this.formularioContacto.get('mail')?.errors?.['email']) {

<span>(El mail no es válido)</span>

}

</p>

<p>Mensaje<br>

<textarea rows="10" cols="70" formControlName="mensaje"></textarea>

</p>

@if(this.formularioContacto.get('mensaje')?.errors?.['required']) {

<div>(El mensaje no puede quedar vacío)</div>

}

@if(this.formularioContacto.get('mensaje')?.errors?.['maxlength']) {

<div>(Debe tener como máximo 500 caracteres)</div>

}

<button type="submit">Confirmar</button>

</form>

<div>{{resultado}}</div>

<router-outlet />

Analizaremos este archivo en conjunto luego de presentar 'app.component.ts'

La clase asociada a la vista es (app.component.ts):

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

import { RouterOutlet } from '@angular/router';

import { ReactiveFormsModule, Validators, FormBuilder, FormGroup } from '@angular/forms';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ReactiveFormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

resultado!: string;

formularioContacto: FormGroup;

constructor(private fb: FormBuilder) {

this.formularioContacto = this.fb.group({

nombre: ['', [Validators.required, Validators.minLength(10)]],

mail: ['', [Validators.required, Validators.email]],

mensaje: ['', [Validators.required, Validators.maxLength(500)]]

});

}

submit() {

if (this.formularioContacto.valid)

this.resultado = "Todos los datos son válidos";

else

this.resultado = "Hay datos inválidos en el formulario";

}

}

Lo primero que debemos hacer es importar la clase FormBuilder (ya no requerimos importar las clases FormGroup y FormControl):

import { ReactiveFormsModule, Validators, FormBuilder, FormGroup } from '@angular/forms';

Se inyecta al constructor un objeto de la clase FormBuilder:

constructor(private fb: FormBuilder) {

}

Llamamos al método 'group' de la clase FormBuilder y le pasamos un objeto literal indicando como atributos los nombres de los controles visuales definidos en HTML y como valor se le pasa un arreglo, cuyo primer elemento es el valor inicial del control y en el segundo elemento del arreglo pasamos otro arreglo con las funciones de validación que se le aplican a dicho campo:

this.formularioContacto = this.fb.group({

nombre: ['', [Validators.required, Validators.minLength(10)]],

mail: ['', [Validators.required, Validators.email]],

mensaje: ['', [Validators.required, Validators.maxLength(500)]]

});

Tener que el resultado es exactamente el mismo si utilizamos la sintaxis vista anteriormente (debemos importar la clase FormControl en dicho caso):

this.formularioContacto = new FormGroup({

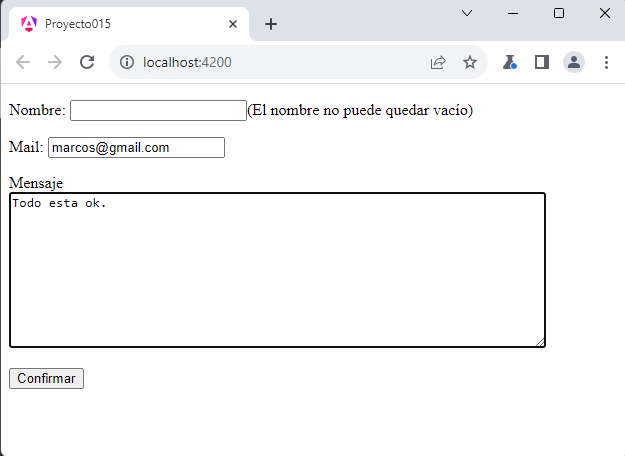
nombre: new FormControl('', [Validators.required, Validators.minLength(10)]),

mail: new FormControl('', [Validators.required, Validators.email]),

mensaje: new FormControl('', [Validators.required, Validators.maxLength(500)])

});

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto015/).

1. Router : definición de rutas

Angular incluye un módulo especial si queremos administrar nuestra aplicación mediante rutas.

Las rutas están dadas por la url, ejemplos de rutas pueden ser:

http://localhost:4200/contacto

http://localhost:4200/acercade

http://localhost:4200/proveedores

Luego según la ruta especificada mostramos una vista distinta, generalmente indicando un nombre de componente.

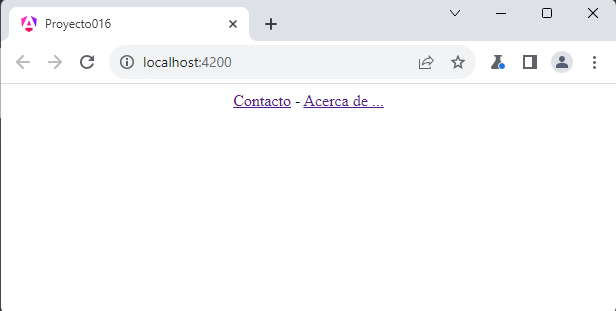
Cuando creamos la aplicación con Angular CLI nos genera los archivos básicos para trabajar con rutas (Router).

Problema

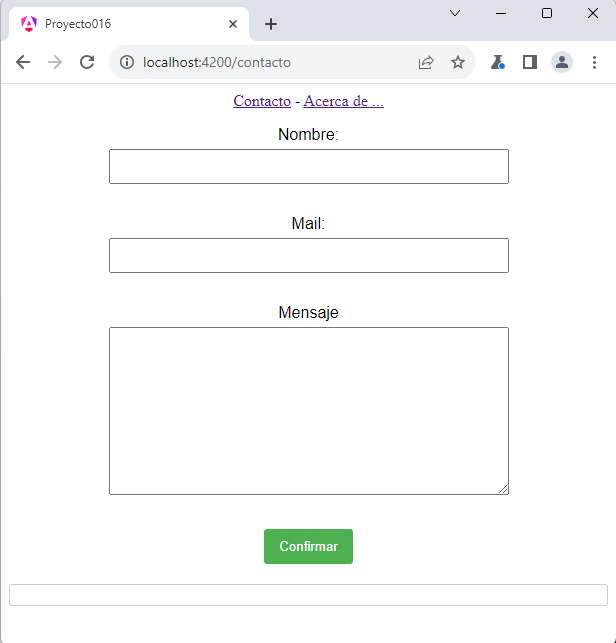
Implementar una aplicación que muestre dos enlaces en la parte superior de la página que acceda el primero a un formulario de contacto y el segundo una página de Acerca de.

Es decir que tenemos las siguientes pantallas:

Cuando se inicia:



Cuando se selecciona el enlace de contacto:



Cuando se selecciona el enlace de Acerca de...:



1. Desde la línea de comandos de Node.js procedemos a crear el proyecto016:
2. ng new proyecto016

Lo primero que vamos a hacer es crear las dos componentes que a posteriori configuraremos una ruta para cada uno.

1. Creamos la componente contacto:
2. ng generate component contacto

Implementamos el siguiente código para los tres archivos:

contacto.component.html

<form [formGroup]="formularioContacto" (ngSubmit)="submit()">

<p>Nombre:

<input type="text" formControlName="nombre">

</p>

<p>Mail:

<input type="text" formControlName="mail">

</p>

<p>Mensaje<br>

<textarea rows="10" cols="70" formControlName="mensaje"></textarea>

</p>

<button type="submit">Confirmar</button>

</form>

<div>{{datos}}</div>

contacto.component.ts

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

import { ReactiveFormsModule, FormControl, FormGroup } from '@angular/forms';

@Component({

selector: 'app-contacto',

imports: [ReactiveFormsModule],

templateUrl: './contacto.component.html',

styleUrl: './contacto.component.css'

})

export class ContactoComponent {

datos = '';

formularioContacto = new FormGroup({

nombre: new FormControl(''),

mail: new FormControl(''),

mensaje: new FormControl('')

});

submit() {

this.datos = `Nombre=${this.formularioContacto.value.nombre}

Mail=${this.formularioContacto.value.mail}

Mensaje=${this.formularioContacto.value.mensaje}

`;

}

}

contacto.component.css

/\* Estilos para el formulario \*/

form {

max-width: 400px;

margin: 0 auto;

font-family: 'Arial', sans-serif;

}

p {

margin-bottom: 15px;

}

input,

textarea {

width: 100%;

padding: 8px;

margin: 5px 0 15px 0;

box-sizing: border-box;

}

button {

background-color: #4caf50;

color: #fff;

padding: 10px 15px;

border: none;

border-radius: 3px;

cursor: pointer;

}

button:hover {

background-color: #45a049;

}

/\* Estilos para el resultado \*/

div {

margin-top: 20px;

padding: 10px;

border: 1px solid #ccc;

border-radius: 3px;

font-weight: bold;

}

Creamos la componente acercade:

ng generate component acercade

Implementamos el siguiente código para los tres archivos:

acercade.component.html

<h1>Programa: xxxxxxxxx</h1>

<p>Desarrollado: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</p>

<p>Fecha: xxxxxxxx</p>

Los otros dos archivos los dejamos sin cambios: 'acercade.component.css' y 'acercade.component.ts'

Ahora si vamos a lo nuevo, abrimos y modificamos el archivo 'app.routes.ts':

import { Routes } from '@angular/router';

import { ContactoComponent } from './contacto/contacto.component';

import { AcercadeComponent } from './acercade/acercade.component';

export const routes: Routes = [

{

path: 'contacto',

component: ContactoComponent

},

{

path: 'acercade',

component: AcercadeComponent

}

];

Modificamos el arreglo 'routes' con una serie de objetos que especifican las rutas. La propiedad path indica el nombre de la ruta y la propiedad component indica el nombre de la componente que se debe dibujar para dicha ruta.

Es importante importar las componentes:

import { ContactoComponent } from './contacto/contacto.component';

import { AcercadeComponent } from './acercade/acercade.component';

Nos queda modificar la componente principal, por un lado el archivo 'app.component.html':

<div style="text-align:center">

<a routerLink="/contacto">Contacto</a> -

<a routerLink="/acercade">Acerca de ...</a>

<router-outlet />

</div>

Mediante la etiqueta 'router-outlet' indicamos el lugar que debe mostrar la componente especificada por la ruta configurada en el archivo 'app.routes.ts' (ahora vemos el objetivo de la etiqueta <router-outlet /> que la dispone Angular por defecto cada vez que creamos un proyecto, en proyectos medianos o grandes es obligatorio el empleo de rutas)

Para cambiar de ruta mediante hipervínculos debemos iniciar la propiedad 'routerLink' asignando la ruta respectiva.

Finalmente el archivo 'app.component.ts':

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

import { RouterOutlet, RouterLink } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet, RouterLink],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

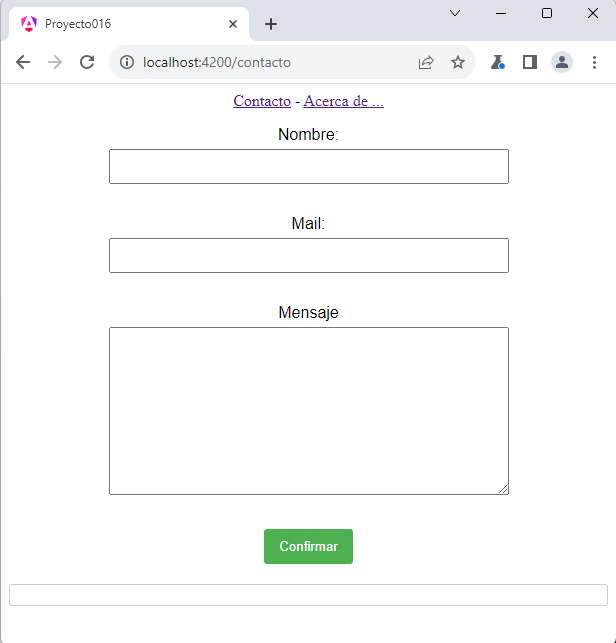
})

export class AppComponent {

}

Importamos la clase 'RouterLink'.

Cuando ejecutemos si accedemos al primer hipervínculo (tener en cuenta que no se recarga la página, no hay una petición al servidor, sino se resuelve la ruta con aplicación Angular en el navegador):



Por otro lado si accedemos a la otra ruta:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto016/).

1. Router : parámetros en una ruta

Angular se base en el concepto de SPA (Single Page Applications - Aplicaciones de una sola página), lo cual no significa que la URL del navegador sea siempre la misma.

El objetivo de Angular es que cuando cambiamos de URL no se recupere la estructura de la página del servidor, sino que se recree en forma local (salvo que se tengan que hacer peticiones de datos dinámicos al servidor)

Vimos que las rutas están dadas por las URL:

http://localhost:4200/

http://localhost:4200/presentacion

http://localhost:4200/articulos

http://localhost:4200/contacto

http://localhost:4200/acercade

Tenemos la ruta raiz '/', otra ruta es 'acercade' etc.

Ahora veremos como trabajar en Angular si una ruta tiene uno o más parámetros. Un ejemplo de ruta con parámetros podría ser pasar el número de artículo en la ruta:

http://localhost:4200/articulos/323

Estamos pasando a la ruta 'articulos' el parámetro '323'. Como vimos anteriormente es obligatorio definir exactamente las rutas que puede procesar la aplicación Angular.

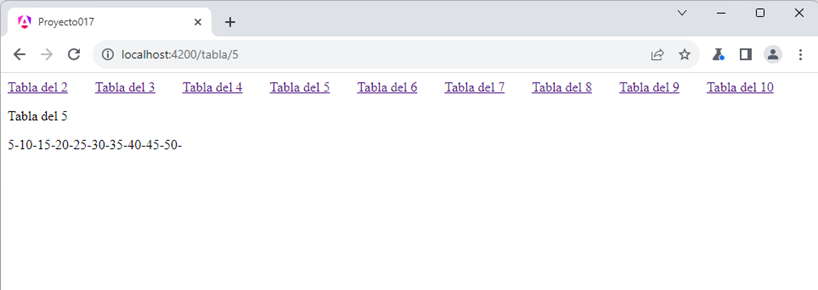
Con un ejemplo analizaremos como declaramos las rutas, pasamos parámetros y como los recuperamos.

Problema

Definir en la componente principal de la aplicación 10 hipervínculos con los números del 1 al 10, que llamen a otra ruta de la aplicación que tenga por objetivo mostrar la tabla de multiplicar de dicho valor.

Crear una segunda componente que reciba el parámetro y proceda a mostrar la tabla de multiplicar.

Los enlaces de los números del 1 al 10 siempre deben quedar visibles, lo que debe actualizarse es la componente que calcula la tabla de multiplicar:



Crearemos primero el proyecto :

ng new proyecto017

Procedemos a crear la componente 'TablanroComponent':

ng generate component tablanro

Modificamos el archivo 'tablanro.component.ts':

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

import { ActivatedRoute, ParamMap } from '@angular/router';

@Component({

selector: 'app-tablanro',

imports: [],

templateUrl: './tablanro.component.html',

styleUrl: './tablanro.component.css'

})

export class TablanroComponent {

nro=0;

tabla='';

constructor(private activatedRoute: ActivatedRoute) {

this.activatedRoute.paramMap.subscribe((parametros: ParamMap) => {

this.nro = parseInt(parametros.get("nro")!);

this.tabla = '';

for (let x = 1; x <= 10; x++) {

let t = x \* this.nro;

this.tabla += t + '-';

}

})

}

}

Importamos la clase ActivedRoute y la interface ParamMap:

import { ActivatedRoute, ParamMap } from '@angular/router';

Al constructor inyectamos un objeto de tipo ActivedRoute:

constructor(private activatedRoute: ActivatedRoute) {

Definimos dos atributos 'nro' y 'tabla' que luego mediante interpolación mostraremos en la vista de la componente:

nro=0;

tabla='';

El método constructor se ejecutará una única vez cuando se crea la componente, luego como el usuario puede presionar sucesivas veces los enlaces con la misma ruta pero con distintos valores en su parámetro, debemos tener alguna forma para que dichos valores se refresquen en la vista. La clase 'ActivatedRoute' dispone de un método llamado 'paramMap' que llamando a suscribe logramos que cada vez que cambia el valor del parámetro de la ruta se ejecute la función anónima que le pasamos como parámetro.

La función anónima recibe como parámetro un objeto de la clase ParamMap que contiene entre otros un método llamado 'get' que nos permite recuperar el parámetro 'nro' de nuestro problema:

this.activatedRoute.paramMap.subscribe((parametros: ParamMap) => {

this.nro = parseInt(parametros.get("nro")!);

this.tabla = '';

for (let x = 1; x <= 10; x++) {

let t = x \* this.nro;

this.tabla += t + '-';

}

})

Modificamos el archivo 'tablanro.component.html':

<p>Tabla del {{nro}}</p>

<p>{{tabla}}</p>

En la vista mostramos el atributo 'nro' y el string con la tabla de multiplicar que lo recuperamos del atributo 'tabla'. Recordar que estos dos atributos se modifican en la función anónima que le pasamos al método 'suscribe' del objeto paramMap que almacena el objeto de la clase 'ActivadedRouter' que se le inyectó en el constructor.

Codificamos la componente principal 'app.component.html':

@for(numero of [2,3,4,5,6,7,8,9,10];track $index) {

<span style="margin-right: 2rem;">

<a routerLink="tabla/{{numero}}">Tabla del {{numero}}</a>

</span>

}

<div>

<router-outlet />

</div>

El template de la componente principal siempre está visible, mediante el @for generamos los enlaces:

@for(numero of [2,3,4,5,6,7,8,9,10];track $index) {

<span style="margin-right: 2rem;">

<a routerLink="tabla/{{numero}}">Tabla del {{numero}}</a>

</span>

}

Es importante notar que mediante la propiedad 'routerLink' indicamos la ruta que debemos cargar y el parámetro a pasar:

routerLink="tabla/{{numero}}"

Luego es fundamental la etiqueta 'router-outlet', que es el lugar donde se sustituye con el contenido de la ruta seleccionada de un enlace de la página:

<router-outlet />

De esta forma logramos que siempre se muestren los enlaces y lo que cambia es la ruta que cargamos y se hace en el lugar que especificamos la etiqueta 'router-outlet'.

En la componente principal debemos importar la clase RouterLink:

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

import { RouterOutlet, RouterLink } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet, RouterLink],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

}

También debemos modificar el archivo 'app.routes.ts':

import { Routes } from '@angular/router';

import { TablanroComponent } from './tablanro/tablanro.component';

export const routes: Routes = [{

path: "tabla/:nro",

component: TablanroComponent

}];

Importamos la clase 'TablanroComponent' :

import { TablanroComponent } from './tablanro/tablanro.component';

Agregamos un objeto al arreglo 'routes' iniciando dos atributos: 'path' que indica el nombre de la ruta y en este caso el parámetro 'nro', y en 'component' indicamos el nombre de la componente que se debe mostrar:

export const routes: Routes = [{

path: "tabla/:nro",

component: TablanroComponent

}];

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto017/).

Acotaciones

Una ruta puede tener más de un parámetro, por ejemplo si necesitamos pasar un segundo parámetro en el ejercicio anterior que indique la cantidad de términos de la tabla de multiplicar que se va a mostrar, deberíamos hacer los siguientes cambios:

En el archivo 'app.routes.ts' cambiar el atributo path de la ruta:

export const routes: Routes = [{

path: "tabla/:nro/:terminos",

component: TablanroComponent

}];

En la componente 'tablanro.component.ts' cambiar el for con el parámetro recibido:

constructor(private activatedRoute: ActivatedRoute) {

this.activatedRoute.paramMap.subscribe((parametros: ParamMap) => {

this.nro = parseInt(parametros.get("nro")!);

this.tabla = '';

for (let x = 1; x <= parseInt(parametros.get("terminos")!); x++) {

let t = x \* this.nro;

this.tabla += t + '-';

}

})

}

Finalmente los enlaces en el archivo 'app.component.html' requieren dos parámetros, por ejemplo si queremos mostrar 20 términos de la tabla de multiplicar, el código debe ser:

<a routerLink="tabla/{{numero}}/20">Tabla del {{numero}}</a>

1. Router : captura de ruta no existente

Como vimos en el concepto anterior en el archivo 'app.routes.ts' configuramos en el arreglo 'routes' todas las rutas que dispone nuestra aplicación, ahora veremos que cambios debemos hacer en el archivo para capturar una ruta inexistente e informar de tal situación al usuario.

Problema

Confeccionar una página que muestre una barra de navegación con 4 enlaces: 'Inicio', 'Mapa', 'Contacto' y 'Acerca de..'.

Definir una ruta para cada sección de la página y en el caso de ingresar el usuario una ruta inexistente mostrar tal situación.

* Crearemos primero el proyecto:
* ng new proyecto018
* Creamos 5 componentes, 4 que representan cada una de las secciones del sitio web y una quinta de ruta inexistente (lo que conocemos "404 page not found", tener en cuenta que Angular no hace realmente una petición al servidor, sino que tiene toda la información ya cargada en el navegador)

Creación de las componentes:

ng generate component Inicio

ng generate component Mapa

ng generate component Contacto

ng generate component AcercaDe

ng generate component Pagina404

Modificamos el archivo 'app.routes.ts' especificando las rutas de nuestra aplicación:

import { Routes } from '@angular/router';

import { InicioComponent } from './inicio/inicio.component';

import { MapaComponent } from './mapa/mapa.component';

import { ContactoComponent } from './contacto/contacto.component';

import { AcercaDeComponent } from './acerca-de/acerca-de.component';

import { Pagina404Component } from './pagina404/pagina404.component';

export const routes: Routes = [{

path: '',

component: InicioComponent

},

{

path: 'inicio',

component: InicioComponent

},

{

path: 'mapa',

component: MapaComponent

},

{

path: 'contacto',

component: ContactoComponent

},

{

path: 'acercade',

component: AcercaDeComponent

},

{

path: 'pagina404',

component: Pagina404Component

},

{

path: '\*\*',

redirectTo: 'pagina404'

}];

La última componente del arreglo es la que indicamos en path la cadena '\*\*' y mediante el atributo 'redirectTo' especificamos a que otro path redireccionar:

{

path: '\*\*',

redirectTo: 'pagina404'

}

Es indispensable para el funcionamiento que la captura de ruta inexistente con los "\*\*" sea la última del arreglo.

Si el usuario no carga ninguna ruta, sino directamente el nombre del dominio, luego tenemos que se muestra la componente 'InicioComponent':

{

path: '',

component: InicioComponent

}

* Procedemos a modificar la vista de la componente que crea Angular 'app.component.html' donde creamos la barra de navegación con los enlaces a las distintas rutas de nuestra aplicación:

<div class="menu">

<a routerLink="inicio">Inicio</a>

<a routerLink="mapa">Mapa</a>

<a routerLink="contacto">Contacto</a>

<a routerLink="acercade">Acerca de...</a>

</div>

<div>

<router-outlet />

</div>

Es importante utilizar la propiedad 'routerLink' para indicar cada una de las rutas.

Disponemos la etiqueta 'router-outlet', que sabemos que es el espacio que será sustituido por la componente respectiva definida en el arreglo 'router'.

El archivo 'app.component.ts' queda con el siguiente código (importamos RouterLink):

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

import { RouterOutlet, RouterLink } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet, RouterLink],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'proyecto018';

}

La hoja de estilo para la componente la especificamos en el archivo 'app.component.css' :

.menu {

display: flex;

justify-content: space-around;

background-color: #333;

padding: 10px;

}

.menu a {

color: #fff;

text-decoration: none;

padding: 5px 10px;

}

.menu a:hover {

background-color: #555;

}

Finalmente agregamos algunos datos a cada una de las 5 componentes que representas las secciones de nuestro sitio web:

inicio.component.html

<h1>Contenido de página de inicio.</h1>

mapa.component.html

<h1>Contenido de página de mapa.</h1>

contacto.component.html

<h1>Contenido de página de contacto.</h1>

acercade.component.html

<h1>Contenido de página de acerca de.</h1>

pagina404.component.html

<h1>Página no encontrada.</h1>

Con los pasos dados ya tenemos nuestro sitio que muestra correctamente cada una de las rutas definidas para nuestro sitio web:



Si ingresa una ruta no válida, por ejemplo "http://localhost:4200/noticias", se procede a redireccionar y mostrar la componente que hemos estipulado nosotros:



Es nuestra decisión a que página redireccionar, podemos inclusive no informar nada y redireccionarlo a la página principal del sitio.

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto018/).

1. Router : rutas anidadas

En un comienzo trabajamos el desarrollo de aplicaciones Angular componiendo la misma como un conjunto de componentes todas en la misma ruta, por ejemplo:

https://www.sitio.com/

Luego vimos que cuando la aplicación comienza a ser más compleja es más conveniente agrupar sus funcionalidades en distintas rutas, con o sin parámetros:

https://www.sitio.com/inicio

https://www.sitio.com/clientes

https://www.sitio.com/contacto

https://www.sitio.com/articulos

Esta forma de organizar las rutas nos permite crear un sitio más grande, pero ahora veremos que podemos crear rutas anidadas, por ejemplo:

https://www.sitio.com/inicio

https://www.sitio.com/clientes/mayoristas

https://www.sitio.com/clientes/minoristas

Veremos que Angular requiere una sintaxis específica cuando definimos el arreglo Routes para indicar que se trata de una ruta anidada.

Problema

Confeccionar una aplicación que muestre una barra de navegación con 2 rutas: 'America' y 'Europa'.  
Cuando se selecciona 'America', mostrar el mapa de 'América' y una barra de navegación con 3 paises americanos, hacer lo mismo si se elije Europa, emplear rutas anidadas para cada país, de tal forma que se acceda a cada país con la siguiente sintaxis:

https://www.sitio.com

https://www.sitio.com/america

https://www.sitio.com/america/chile

https://www.sitio.com/america/argentina

https://www.sitio.com/america/uruguay

https://www.sitio.com/europa

https://www.sitio.com/europa/espana

https://www.sitio.com/europa/francia

https://www.sitio.com/europa/italia

Crearemos primero el proyecto:

ng new proyecto019

Crearemos 2 componentes para cada continente y 6 componentes más para cada país (si copiamos y pegamos en la consola de VSCode podemos rápidamente crear las 8 componentes):

ng generate component america

ng generate component america/chile

ng generate component america/argentina

ng generate component america/uruguay

ng generate component europa

ng generate component europa/espana

ng generate component europa/francia

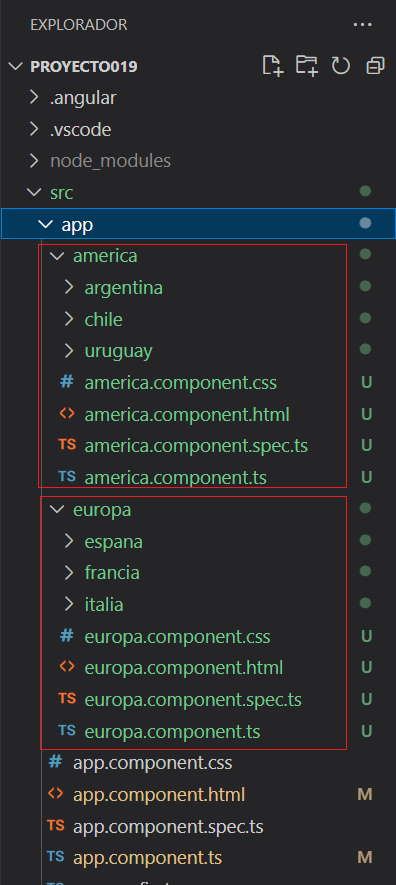
ng generate component europa/Italia

Podíamos crear todas las componentes en la misma altura y no crear subcarpetas para cada país, pero queda mucho más ordenada nuestra aplicación si comenzamos a agrupar las componentes en distintas carpetas y subcarpetas en forma jerárquica. Angular CLI crea automáticamente una subcarpeta cuando no existe al utilizar la sintaxis:

ng generate component america/chile

Se crea la carpeta 'america' y dentro de la misma se guardan los cuatro archivos asociados a la componente 'chile'.

Podemos ver en el editor de texto las 8 componentes creadas:



El caracter 'ñ' no se puede utilizar como selector: **Selector (app-españa) is invalid.**

Procedemos a modificar el archivo 'app.routes.ts' definiendo las rutas principales y las anidadas:

import { Routes } from '@angular/router';

import { AmericaComponent } from './america/america.component';

import { ChileComponent } from './america/chile/chile.component';

import { ArgentinaComponent } from './america/argentina/argentina.component';

import { UruguayComponent } from './america/uruguay/uruguay.component';

import { EuropaComponent } from './europa/europa.component';

import { EspanaComponent } from './europa/espana/espana.component';

import { FranciaComponent } from './europa/francia/francia.component';

import { ItaliaComponent } from './europa/italia/italia.component';

export const routes: Routes = [

{

path: 'america',

component: AmericaComponent,

children: [

{

path: 'chile',

component: ChileComponent

},

{

path: 'argentina',

component: ArgentinaComponent

},

{

path: 'uruguay',

component: UruguayComponent

}

]

},

{

path: 'europa',

component: EuropaComponent,

children: [

{

path: 'espana',

component: EspanaComponent

},

{

path: 'francia',

component: FranciaComponent

},

{

path: 'italia',

component: ItaliaComponent

}

]

}

];

Agregamos en cada objeto que tiene rutas anidadas la propiedad 'children', la cual es un arreglo de objetos donde definimos todas las rutas anidadas.

Luego podemos indicar en el navegador una ruta principal:

https://www.sitio.com/america

Pero también podemos acceder a una ruta anidada con la sintaxis:

https://www.sitio.com/america/chile

Procedemos a modificar la componente principal que nos genera Angular 'app.component.html':

<div class="menu">

<a routerLink="america">America</a>

<a routerLink="europa">Europa</a>

</div>

<div>

<router-outlet />

</div>

La directiva 'router-outlet' es indispensable para que se muestren alguna de las dos componentes 'AmericaComponent' o 'EuropaComponent' según que enlace se seleccione.

También modificamos el archivo app.component.ts:

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

import { RouterOutlet, RouterLink } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet, RouterLink],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

title = 'proyecto019';

}

Cuando se selecciona la ruta 'america' se carga la componentes 'AmericaComponent', modificamos la vista de dicha componente 'america.component.html'

<iframe

src="https://www.google.com/maps/embed?pb=!1m14!1m12!1m3!1d31536306.714862086!2d-54.01447990393665!3d-20.38062741924525!2m3!1f0!2f0!3f0!3m2!1i1024!2i768!4f13.1!5e0!3m2!1ses!2sar!4v1583676893900!5m2!1ses!2sar"

width="600" height="450" frameborder="0" style="border:0;" allowfullscreen=""></iframe>

<div class="menu">

<a routerLink="chile">Chile</a>

<a routerLink="argentina">Argentina</a>

<a routerLink="uruguay">Uruguay</a>

</div>

<router-outlet />

Disponemos tres enlaces hacia las rutas anidadas, recordemos que esta componente se activa cuando se seleccionó la ruta 'america'.

También esta componente dispone la etiqueta 'router-outlet' donde deben visualizarse las componentes 'ArgentinaComponent', 'ChileComponent' o 'UruguayComponent'.

También debemos modificar el archivo 'america.component.ts' agregando los import respectivos

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

import { RouterOutlet, RouterLink } from '@angular/router';

@Component({

selector: 'app-america',

imports: [RouterOutlet, RouterLink],

templateUrl: './america.component.html',

styleUrl: './america.component.css'

})

export class AmericaComponent {

}

Exactamente lo mismo hacemos para crear la componente 'europa.component.html':

<iframe src="https://www.google.com/maps/embed?pb=!1m14!1m12!1m3!1d9880418.608500127!2d2.640727530399596!3d45.69187171781666!2m3!1f0!2f0!3f0!3m2!1i1024!2i768!4f13.1!5e0!3m2!1ses!2sar!4v1583677509059!5m2!1ses!2sar" width="600" height="450" frameborder="0" style="border:0;" allowfullscreen=""></iframe>

<div class="menu">

<a routerLink="espana">España</a>

<a routerLink="francia">Francia</a>

<a routerLink="italia">Italia</a>

</div>

<router-outlet />

También debemos modificar el archivo 'europa.component.ts' agregando los import respectivos

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

import { RouterOutlet, RouterLink } from '@angular/router';

@Component({

selector: 'app-europa',

imports: [RouterOutlet, RouterLink],

templateUrl: './europa.component.html',

styleUrl: './europa.component.css'

})

export class EuropaComponent {

}

Ya tenemos prácticamente finalizado el ejemplo de rutas anidadas con Angular, podemos modificar las vistas de las componentes de cada país:

chile.component.html

argentina.component.html

uruguay.component.html

espana.component.html

francia.component.html

italia.component.html

Vamos a disponer una serie de estilos comunes a todos los menúes de la aplicación, para ello modificamos el archivo 'styles.css' que se encuentra en la raiz del proyecto:

/\* You can add global styles to this file, and also import other style files \*/

.menu {

display: flex;

justify-content: space-around;

background-color: #333;

padding: 10px;

}

.menu a {

color: #fff;

text-decoration: none;

padding: 5px 10px;

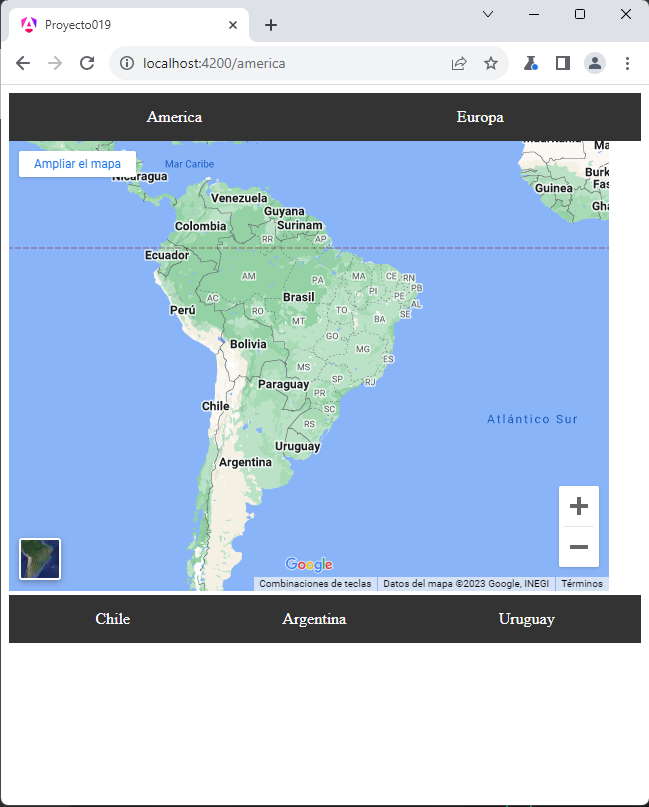
}

.menu a:hover {

background-color: #555;

}

Si seleccionamos la opción de América:



Finalmente si ingresamos a una ruta anidada por ejemplo Argentina:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto019/).

1. Servicios : concepto y pasos para crearlos

El pilar fundamental cuando creamos una aplicación en Angular es la correcta definición de sus componentes y relaciones entre ellas.

La propuesta del framework de Angular es delegar todas las responsabilidades de acceso a datos (peticiones y envío de datos) y lógica de negocios en otras clases que colaboran con las componentes. Estas clases en Angular se las llama servicios.

Tal es la importancia de los servicios en Angular que la herramienta Angular CLI nos provee la capacidad de crearlos.

Veremos con un ejemplo los pasos que debemos dar para crear un servicio y luego consumirlo desde una componente.

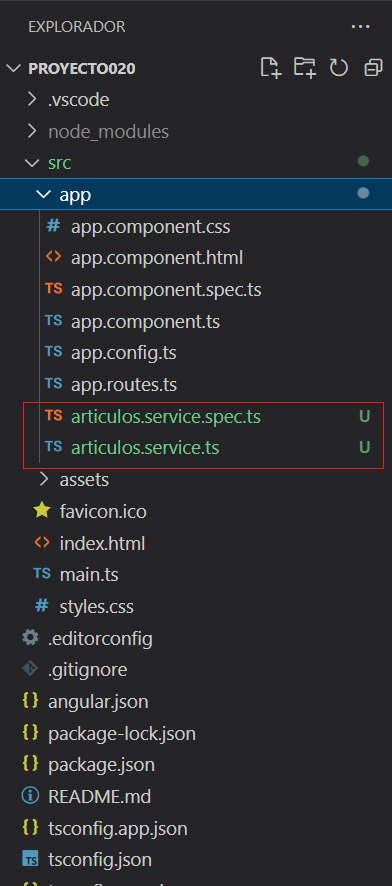
Problema

Confeccionar una aplicación Angular que muestre una lista de artículos. Los artículos almacenarlos en un vector localizado en un servicio. Desde la componente acceder al servicio para pedir los artículos a ser mostrados.

1. Desde la línea de comandos de Node.js procedemos a crear el proyecto020:
2. ng new proyecto020
3. Crearemos el servicio que contiene en memoria la lista de artículos (en muchos casos como veremos más adelante el servicio tiene la responsabilidad de recuperar los datos de un servidor web):
4. ng generate service articulos

Con el comando anterior estamos creando la clase 'ArticulosService'.

Se crean dos archivos:



El código generado de la clase 'ArticulosService' es:

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

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

constructor() { }

}

Lo modificamos por el siguiente código que permita recuperar desde la componente el vector de artículos:

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

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

constructor() { }

retornar() {

return [

{

codigo: 1,

descripcion: "papas",

precio: 12.33

},

{

codigo: 2,

descripcion: "manzanas",

precio: 54

},

{

codigo: 3,

descripcion: "sandía",

precio: 14

}

];

}

}

El decorador @Injectable() será de suma importancia para poder acceder a esta clase desde la componente.

Ahora veremos como consumimos el servicio desde nuestra componente. Procedemos a modificar la componente que se crea por defecto 'AppComponent' que tiene por responsabilidad mostrar en la página el listado de artículos:

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

import { RouterOutlet } from '@angular/router';

import { ArticulosService } from './articulos.service';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

articulos :any;

constructor(private articulosServicio: ArticulosService) {

this.articulos=this.articulosServicio.retornar();

}

}

Primero importamos el servicio llamado ArticulosService que se almacena en el archivo 'articulos.service.ts':

import { ArticulosService } from './articulos.service';

Para inyectar el objeto de la clase 'ArticulosService' que crea Angular en forma automática lo hacemos en el parámetro del constructor::

constructor(private articulosServicio: ArticulosService) {

this.articulos=this.articulosServicio.retornar();

}

Esta asignación dispara la actualización de la página HTML.

1. Falta que codifiquemos la vista con los datos recuperados:

app.component.html

<table>

@for(articulo of articulos;track $index) {

<tr>

<td>{{articulo.codigo}}</td>

<td>{{articulo.descripcion}}</td>

<td>{{articulo.precio}}</td>

</tr>

}

</table>

<router-outlet />

1. Por último la hoja de estilo de la componente:

app.component.css

/\* Estilo para la tabla \*/

table {

border-collapse: collapse;

width: 100%;

}

/\* Estilo para las celdas de la tabla \*/

td {

border: 1px solid #dddddd;

text-align: left;

padding: 8px;

}

/\* Estilo para las filas impares \*/

tr:nth-child(odd) {

background-color: #f2f2f2;

}

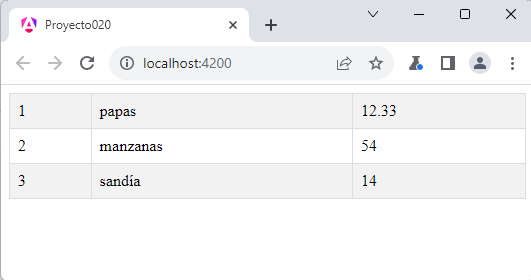
/\* Estilo para las filas al pasar el ratón sobre ellas \*/

tr:hover {

background-color: #e6e6e6;

}

1. Si ejecutamos ahora el proyecto020 veremos en el navegador el listado de artículos:
2. ng server -o



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto020/).

Al principio y con problemas muy sencillos parece que solo agregamos complejidad a nuestra aplicación.

Veremos que esta forma de desacoplar el acceso a datos de las componentes y delegarla en otras clases llamadas servicios facilita el mantenimiento de nuestras aplicaciones.

También hay que hacer notar que la forma de consumir dichas clases se hace por medio del patrón de inyección de dependencias.

1. Servicios : recuperación de datos de un servidor web

Hemos dicho en el concepto anterior que en Angular se delega todas las responsabilidades de acceso a datos (peticiones y envío de datos) y lógica de negocios en otras clases que colaboran con las componentes y son llamados servicios.

Veremos ahora como recuperar datos de un servidor web implementando dicha actividad en un servicio.

Problema

Confeccionar una aplicación que recupere una respuesta en JSON de la dirección:

https://ejerciciostutorialesya.com/vue/datos.php

La estructura del archivo JSON es:

[

{

"codigo": 1,

"descripcion": "papas",

"precio": 12.33

},

{

"codigo": 2,

"descripcion": "manzanas",

"precio": 54

},

{

"codigo": 3,

"descripcion": "sandia",

"precio": 31

}

]

Mostrar en una tabla HTML todos los artículos recuperados.

La recuperación de datos se debe hacer en un servicio.

1. Desde la línea de comandos de Node.js procedemos a crear el proyecto021:
2. ng new proyecto021
3. Crearemos el servicio que recuperará desde un servidor la lista de artículos
4. ng generate service articulos

Con el comando anterior estamos creando la clase 'ArticulosService'

Se crean dos archivos.

El código generado de la clase 'ArticulosService' es:

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

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

constructor() { }

}

Lo modificamos por el siguiente código que permita recuperar desde un servidor web el archivo JSON:

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

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

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

constructor(private http: HttpClient) { }

retornar() {

return this.http.get("https://ejerciciostutorialesya.com/vue/datos.php");

}

}

Debemos ahora modificar el archivo 'app.config.ts' para poder utilizar la función provideHttpClient:

import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';

import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

import { provideHttpClient } from '@angular/common/http';

import { withFetch } from '@angular/common/http';

export const appConfig: ApplicationConfig = {

providers: [provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes), provideHttpClient(withFetch())]

};

1. Ahora veremos como consumimos el servicio desde nuestra componente. Procedemos a modificar la componente que se crea por defecto 'AppComponent' que tiene por responsabilidad mostrar en la página el listado de artículos:
2. import { Component } from '@angular/core';
3. import { RouterOutlet } from '@angular/router';
4. import { ArticulosService } from './articulos.service';
5. @Component({
6. selector: 'app-root',
7. imports: [RouterOutlet],
8. templateUrl: './app.component.html',
9. styleUrls: ['./app.component.css']
10. })
11. export class AppComponent {
12. articulos: any;
13. constructor(private articulosService: ArticulosService) {
14. this.articulosService.retornar()
15. .subscribe(result => this.articulos = result)
16. }
17. }

Primero importamos el servicio llamado ArticulosService que se almacena en el archivo 'articulos.service.ts':

import { ArticulosService } from './articulos.service';

Para inyectar el objeto de la clase 'ArticulosService' que crea Angular en forma automática lo hacemos en el parámetro del constructor:

constructor(private articulosService: ArticulosService) {

this.articulosService.retornar()

.subscribe(result => this.articulos = result)

}

Se almacena en el atributo 'articulosServicio' la referencia del objeto de la clase 'ArticulosService' que crea Angular.

Esta asignación dispara la actualización de la página HTML.

1. Falta que codifiquemos la vista con los datos recuperados:

app.component.html

<table>

<tr>

<td>Codigo</td><td>Descripcion</td><td>Precio</td>

</tr>

@for(art of articulos;track art.codigo) {

<tr>

<td>{{art.codigo}}</td>

<td>{{art.descripcion}}</td>

<td>{{art.precio}}</td>

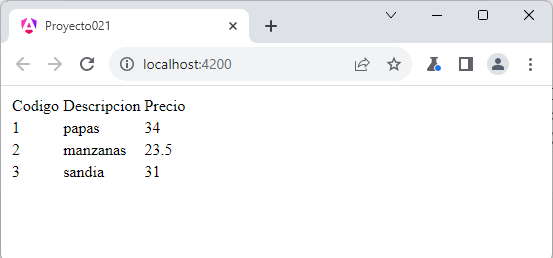
</tr>

}

</table>

<router-outlet />

1. Si ejecutamos ahora el proyecto021 veremos en el navegador el listado de artículos, pero ahora recuperados de un servidor y no extraidos de un vector como en el concepto anterior:
2. ng server -o



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto021/).

1. Comunicación entre Angular y PHP

Hemos dicho que Angular es un framework para el desarrollo de aplicaciones web de una sola página. Una situación muy común es tener que almacenar en un servidor de internet los datos que se ingresan en la aplicación Angular.

Existen muchas tecnologías para procesar los datos que envía y recibe la aplicación Angular, una de la más extendidas en el lenguaje PHP y mediante este acceder a una base de datos MySql.

En este concepto dejaremos en forma muy clara todos los pasos que debemos desarrollar tanto en el cliente (aplicación angular) como en el servidor (aplicación PHP y MySql)

Problema

Confeccionar una aplicación que permita administrar una lista de artículos, cada artículo almacena el código, descripción y precio . Se debe poder agregar, borrar y modificar los datos de un artículo almacenados en una base de datos MySQL y accedida con una programa en PHP.

Desde la línea de comandos de Node.js procedemos a crear el proyecto022:

f:\angulardevya> ng new proyecto022

Recordemos que la propuesta del framework de Angular es delegar todas las responsabilidades de acceso a datos (peticiones y envío de datos) y lógica de negocios en otras clases que colaboran con las componentes. Estas clases en Angular se las llama servicios.

Crearemos el servicio 'articulos' para ello utilizamos Angular CLI:

f:\angulardevya\proyecto022\> ng generate service articulos

Con el comando anterior estamos creando la clase 'ArticulosService'

El código que debemos guardar en el archivo 'articulos.service.ts' es:

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

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

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

url='https://ejerciciostutorialesya.com/angular/proyecto016/'; // disponer url de su servidor que tiene las páginas PHP

constructor(private http: HttpClient) { }

recuperarTodos() {

return this.http.get(`${this.url}recuperartodos.php`);

}

alta(articulo:any) {

return this.http.post(`${this.url}alta.php`, JSON.stringify(articulo));

}

baja(codigo:number) {

return this.http.get(`${this.url}baja.php?codigo=${codigo}`);

}

seleccionar(codigo:number) {

return this.http.get(`${this.url}seleccionar.php?codigo=${codigo}`);

}

modificacion(articulo:any) {

return this.http.post(`${this.url}modificacion.php`, JSON.stringify(articulo));

}

}

Presentaremos primero todos los archivos y luego explicaremos como se relacionan.

Debemos ahora modificar el archivo 'app.config.ts' para poder utilizar la función provideHttpClient:

import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';

import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

import { provideHttpClient } from '@angular/common/http';

import { withFetch } from '@angular/common/http';

export const appConfig: ApplicationConfig = {

providers: [provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes), provideHttpClient(withFetch())]

};

Nuestra única componente debe implementar el siguiente código en el archivo 'app.component.ts':

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

import { RouterOutlet } from '@angular/router';

import { ArticulosService } from './articulos.service';

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

@Component({

selector: 'app-root',

imports: [RouterOutlet, FormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

articulos:any;

art={

codigo:0,

descripcion:"",

precio:0

}

constructor(private articulosServicio: ArticulosService) {

this.recuperarTodos();

}

recuperarTodos() {

this.articulosServicio.recuperarTodos().subscribe((result:any) => this.articulos = result);

}

alta() {

this.articulosServicio.alta(this.art).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

baja(codigo:number) {

this.articulosServicio.baja(codigo).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

modificacion() {

this.articulosServicio.modificacion(this.art).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

seleccionar(codigo:number) {

this.articulosServicio.seleccionar(codigo).subscribe((result:any) => this.art = result[0]);

}

}

Y el archivo 'app.component.html' con:

<div>

<h1>Administración de artículos</h1>

<table border="1">

<tr>

<td>Codigo</td><td>Descripcion</td><td>Precio</td><td>Borrar</td><td>Seleccionar</td>

</tr>

@for(art of articulos;track art.codigo) {

<tr>

<td>{{art.codigo}}</td>

<td>{{art.descripcion}}</td>

<td>{{art.precio}}</td>

<td><button (click)="baja(art.codigo)">Borrar?</button></td>

<td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>

</tr>

}

</table>

<div>

<p>

descripcion:<input type="text" [(ngModel)]="art.descripcion" />

</p>

<p>

precio:<input type="number" [(ngModel)]="art.precio" />

</p>

<p><button (click)="alta()">Agregar</button>

<button (click)="modificacion()">Modificar</button></p>

</div>

</div>

<router-outlet />

Todos los archivos presentados son los necesarios en Angular 19, veamos ahora que tenemos en PHP y MySQL.

Primero debemos crear una base de datos en MySQL llamada 'bd1' y crear la siguiente tabla:

CREATE TABLE articulos (

codigo int AUTO\_INCREMENT,

descripcion varchar(50),

precio float,

PRIMARY KEY (codigo)

)

Tenemos una serie de archivos PHP que reciben datos en formato JSON y retornan también un JSON.

El archivo 'recuperartodos.php' retorna en formato JSON todos los artículos:

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");

$con=retornarConexion();

$registros=mysqli\_query($con,"select codigo, descripcion, precio from articulos");

$vec=[];

while ($reg=mysqli\_fetch\_array($registros))

{

$vec[]=$reg;

}

$cad=json\_encode($vec);

echo $cad;

header('Content-Type: application/json');

?>

El archivo 'alta.php' recibe los datos en formato JSON y los almacena en la tabla:

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

$json = file\_get\_contents('php://input');

$params = json\_decode($json);

require("conexion.php");

$con=retornarConexion();

mysqli\_query($con,"insert into articulos(descripcion,precio) values

('$params->descripcion',$params->precio)");

class Result {}

$response = new Result();

$response->resultado = 'OK';

$response->mensaje = 'datos grabados';

header('Content-Type: application/json');

echo json\_encode($response);

?>

El archivo 'baja.php':

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");

$con=retornarConexion();

mysqli\_query($con,"delete from articulos where codigo=$\_GET[codigo]");

class Result {}

$response = new Result();

$response->resultado = 'OK';

$response->mensaje = 'articulo borrado';

header('Content-Type: application/json');

echo json\_encode($response);

?>

El archivo 'modificacion.php':

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

$json = file\_get\_contents('php://input');

$params = json\_decode($json);

require("conexion.php");

$con=retornarConexion();

mysqli\_query($con,"update articulos set descripcion='$params->descripcion',

precio=$params->precio

where codigo=$params->codigo");

class Result {}

$response = new Result();

$response->resultado = 'OK';

$response->mensaje = 'datos modificados';

header('Content-Type: application/json');

echo json\_encode($response);

?>

El archivo 'seleccionar.php':

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");

$con=retornarConexion();

$registros=mysqli\_query($con,"select codigo, descripcion, precio from articulos where codigo=$\_GET[codigo]");

if ($reg=mysqli\_fetch\_array($registros))

{

$vec[]=$reg;

}

$cad=json\_encode($vec);

echo $cad;

header('Content-Type: application/json');

?>

El archivo 'conexion.php':

<?php

function retornarConexion() {

$con=mysqli\_connect("localhost","root","","bd1");

return $con;

}

?>

El resultado de ejecutar esta aplicación en el navegador, teniendo en funcionamiento el servidor con PHP y MySQL es:



Explicación

Ahora veremos como funciona esta aplicación cliente/servidor implementada con Angular en el cliente y PHP en el servidor.

* Recuperación de todos los registros

Inmediatamente se inicia la aplicación Angular se crea el objeto de la clase 'AppComponent' (nuestra única componente), en esta clase debe llegar al constructor el objeto de la clase 'ArticulosService' y procedemos a llamar al método 'recuperarTodos':

constructor(private articulosServicio: ArticulosService) {

this.recuperarTodos();

}

La clase ArticulosService está creada en el archivo 'articulos.service.ts':

export class ArticulosService {

...

}

recuperarTodos tiene por objetivo utilizar el 'servicio' que llega al constructor para llamar al método 'recuperarTodos' del servicio propiamente dicho:

recuperarTodos() {

this.articulosServicio.recuperarTodos().subscribe((result:any) => this.articulos = result);

}

Si vemos ahora el método 'recuperarTodos' de la clase 'ArticulosService', es el que tiene la responsabilidad de hacer una petición al servidor:

recuperarTodos() {

return this.http.get(`${this.url}recuperartodos.php`);

}

El método 'recuperarTodos' de la clase 'ArticulosService' retorna un objeto de la clase 'HttpClient'.

Ahora debemos entender porque podemos llamar al método 'suscribe':

recuperarTodos() {

this.articulosServicio.recuperarTodos().subscribe((result:any) => this.articulos = result);

}

El método 'suscribe' recibe los resultados y procedemos a asignar a la propiedad 'articulos', con esto, Angular se encarga de actualizar la página con todos los artículos recuperados. El proceso de actualizar la página sucede en el archivo 'app.component.html':

<div>

<h1>Administración de artículos</h1>

<table border="1">

<tr>

<td>Codigo</td><td>Descripcion</td><td>Precio</td><td>Borrar</td><td>Seleccionar</td>

</tr>

@for(art of articulos;track art.codigo) {

<tr>

<td>{{art.codigo}}</td>

<td>{{art.descripcion}}</td>

<td>{{art.precio}}</td>

<td><button (click)="baja(art.codigo)">Borrar?</button></td>

<td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>

</tr>

}

</table>

<div>

<p>

descripcion:<input type="text" [(ngModel)]="art.descripcion" />

</p>

<p>

precio:<input type="number" [(ngModel)]="art.precio" />

</p>

<p><button (click)="alta()">Agregar</button>

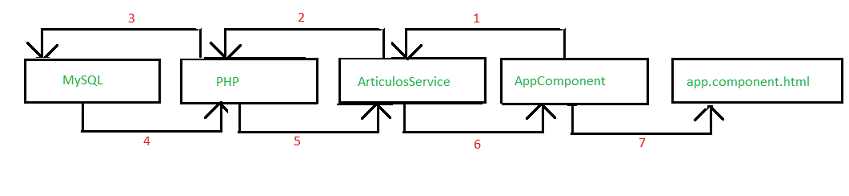
<button (click)="modificacion()">Modificar</button></p>

</div>

</div>

<router-outlet />

El flujo de la información lo podemos representar con el siguiente esquema:



* Alta

Veamos ahora los pasos cuando se agrega una fila en la tabla 'articulos'.  
Todo comienza cuando el operador presiona el botón de 'Agregar':



La etiqueta button tiene enlazado la llamada al método 'alta':

<p><button (click)="alta()">Agregar</button>

El método 'alta' se encuentra codificado en el archivo 'app.component.ts' dentro de la clase 'AppComponent':

alta() {

this.articulosServicio.alta(this.art).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

En este método procedemos a llamar al método 'alta' de la clase 'ArticulosService' y se le pasa como parámetro la propiedad 'art' que almacena la descripción y precio del artículo que el operador acaba de ingresar por teclado.

El método 'alta' de la clase 'ArticulosService' hace la llamada al servidor mediante el objeto 'http' de la clase 'HttpClient'. Se utiliza el método 'post' ya que se enviarán datos al servidor:

alta(articulo:any) {

return this.http.post(`${this.url}alta.php`, JSON.stringify(articulo));

}

Ahora se ejecuta el programa PHP definido en el archivo 'alta.php' donde procedemos a efectuar el insert en la tabla de MySQL:

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

$json = file\_get\_contents('php://input');

$params = json\_decode($json);

require("conexion.php");

$con=retornarConexion();

mysqli\_query($con,"insert into articulos(descripcion,precio) values

('$params->descripcion',$params->precio)");

class Result {}

$response = new Result();

$response->resultado = 'OK';

$response->mensaje = 'datos grabados';

header('Content-Type: application/json');

echo json\_encode($response);

?>

También dentro del programa PHP procedemos a retornar en formato JSON que la operación se efectuó en forma correcta.

En el método 'alta' de la clase 'AppComponent' que ya vimos, recibe los datos de la respuesta JSon, mostrando un mensaje si la carga se efectuó correctamente:

alta() {

this.articulosServicio.alta(this.art).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

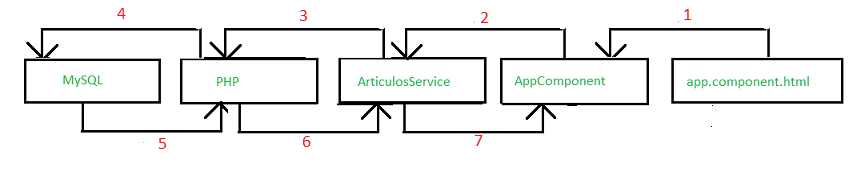
}

});

}

También llamamos al método 'recuperarTodos' con el objetivo que se actualice la pantalla con los datos actuales de la tabla 'articulos'.

El flujo de la información para efectuar el 'alta' de un articulo en la base de datos es::



La actualización de la página HTML la logramos llamando al método 'recuperarTodos'.

* Baja

El borrado de un artículo se efectúa cuando el operador presiona el botón con la etiqueta 'Borra?':

<td><button (click)="baja(art.codigo)">Borrar?</button></td>

Se llama al método 'baja' de la clase 'AppComponent' y se le pasa como parámetro el código de artículo a borrar.

El método baja:

baja(codigo:number) {

this.articulosServicio.baja(codigo).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

Llmamos al método 'baja' de la clase 'ArticulosService' y le pasamos como parámetro el código de artículo que queremos borrar.

El método 'baja' de la clase 'ArticulosService' procede a llamar al archivo baja.php y le pasa como parámetro el código de artículo que se debe borrar:

baja(codigo:number) {

return this.http.get(`${this.url}baja.php?codigo=${codigo}`);

}

Ahora en el servidor se ejecuta la aplicación PHP baja.php:

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");

$con=retornarConexion();

mysqli\_query($con,"delete from articulos where codigo=$\_GET[codigo]");

class Result {}

$response = new Result();

$response->resultado = 'OK';

$response->mensaje = 'articulo borrado';

header('Content-Type: application/json');

echo json\_encode($response);

?>

Luego en el método 'baja' de la clase 'AppComponent' podemos mostrar un mensaje si la baja se ejecutó con éxito:

baja(codigo:number) {

this.articulosServicio.baja(codigo).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

Como podemos ver para actualizar la pantalla procedemos a llamar al método 'recuperarTodos'.

* Consulta

La consulta o selección se dispara cuando el operador presiona el botón que tiene la etiqueta 'Seleccionar' y tiene por objetivo mostrar en los dos controles 'text' la descripción y precio del artículo:

<td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>

Al presionar el botón se llama el método 'seleccionar' de la clase 'AppComponent':

seleccionar(codigo:number) {

this.articulosServicio.seleccionar(codigo).subscribe((result:any) => this.art = result[0]);

}

Ahora llamamos al método 'seleccionar' de la clase 'ArticulosService':

seleccionar(codigo:number) {

return this.http.get(`${this.url}seleccionar.php?codigo=${codigo}`);

}

Recuperamos del servidor llamando a la página 'seleccionar.php' los datos del artículo cuyo código pasamos como parámetro :

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");

$con=retornarConexion();

$registros=mysqli\_query($con,"select codigo, descripcion, precio from articulos where codigo=$\_GET[codigo]");

if ($reg=mysqli\_fetch\_array($registros))

{

$vec[]=$reg;

}

$cad=json\_encode($vec);

echo $cad;

header('Content-Type: application/json');

?>

La ejecución del programa en PHP procede a recuperar la fila de la tabla que coincide con el código enviado y lo retorna con formato JSON.

En el método 'seleccionar' de la clase AppComponent al ejecutarse el método subscribe almacena en la propiedad 'art' el resultado devuelto por el servidor (con esta asignación se actualizan los dos controles 'input' del HTML):

seleccionar(codigo:number) {

this.articulosServicio.seleccionar(codigo).subscribe((result:any) => this.art = result[0]);

}

* Modificación

El último algoritmo que implementa nuestra aplicación es la modificación de la descripción y precio de un artículo que seleccionemos primeramente.

Cuando presionamos el botón que tiene la etiqueta 'Modificar' se ejecuta el método 'modificación':

<button (click)="modificacion()">Modificar</button></p>

El método 'modificación' se implementa en la clase 'AppComponent':

modificacion() {

this.articulosServicio.modificacion(this.art).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

Lo primero que hacemos en este método es llamar al método 'modificación' de la clase 'ArticulosService' y pasar como dato todos los datos del artículo seleccionado y las posibles modificaciones efectuadas.

El método 'modificacion' de la clase 'ArticulosService':

modificacion(articulo:any) {

return this.http.post(`${this.url}modificacion.php`, JSON.stringify(articulo));

}

Accede al servidor pidiendo el archivo 'modificacion.php' y pasando todos los datos del artículo mediante un 'post'.

El archivo 'modificacion.php' procede a cambiar la descripción y precio del artículo:

<?php

header('Access-Control-Allow-Origin: \*');

header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

$json = file\_get\_contents('php://input');

$params = json\_decode($json);

require("conexion.php");

$con=retornarConexion();

mysqli\_query($con,"update articulos set descripcion='$params->descripcion',

precio=$params->precio

where codigo=$params->codigo");

class Result {}

$response = new Result();

$response->resultado = 'OK';

$response->mensaje = 'datos modificados';

header('Content-Type: application/json');

echo json\_encode($response);

?>

En la clase 'AppComponent' podemos comprobar si el resultado fue 'OK' y actualizar nuevamente la página:

modificacion() {

this.articulosServicio.modificacion(this.art).subscribe((datos:any) => {

if (datos['resultado']=='OK') {

alert(datos['mensaje']);

this.recuperarTodos();

}

});

}

Hemos explicado con este problema todos los pasos esenciales para implementar una aplicación en Angular que se comunica con un servidor web con PHP y acceder a una base de datos MySQL.

Puede probar funcionando esta aplicación de Angular desde [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto022/)

1. Pipes: definición

Las pipes también llamadas tuberías o filtros son funciones que se llaman en una vista (html) y que tienen por objetivo transformar un dato a mostrar para mejorar la experiencia del usuario.

Hay una cantidad reducida de pipes que trae por defecto Angular y podemos hacer uso en cualquier vistas de nuestras componentes. Lo más importante es que podemos crear nuestras propias pipes, lo veremos en el próximo concepto.

La llamada a estas funciones tiene una sintaxis muy distinta a la tradicional y su objetivo es hacer más claro la sintaxis de nuestra vista.

Por ejemplo para mostrar el contenido de una variable toda en mayúsculas en la plantilla utilizamos la siguiente sintaxis:

<p>Nombre del cliente:{{ nombre | uppercase}}</p>

Considerando que en la componente hemos definido la propiedad:

nombre = 'Juan Carlos';

La salida en el navegador será:

Nombre del cliente:JUAN CARLOS

Como vemos en la interpolación de la propiedad 'nombre' llamamos a la pipe después del caracter '|'.

La pipe 'uppercase' transforma a mayúsculas el dato que se muestra en la página, no se modifica la propiedad 'nombre'.

En algunas pipes podemos pasar parámetros:

<p>El saldo es:{{ saldo | currency:'$'}}</p>

Seguido al nombre de la pipe disponemos cada parámetro antecedido por el caracter ':'.

Podemos aplicar una pipe directamente a un valor indicado en la vista:

<p>{{ 'Hola' | uppercase }}</p>

Como se trata de un string debemos indicarlo entre comillas. Luego cuando ejecutamos en el navegador tenemos como resultado 'HOLA'.

Problema

Confeccionar una aplicación que defina una serie de propiedades y muestre sus valores aplicando distintas pipes que trae por defecto Angular.

1. Desde la línea de comandos de Node.js procedemos a crear el proyecto023:
2. f:\angularya> ng new proyecto023
3. Definimos ahora en la componente 'AppComponent' una serie de propiedades:
4. import { Component } from '@angular/core';
5. import { RouterOutlet } from '@angular/router';
6. import { UpperCasePipe, LowerCasePipe, CurrencyPipe, DatePipe, JsonPipe, SlicePipe } from '@angular/common';
7. @Component({
8. selector: 'app-root',
9. imports: [RouterOutlet, UpperCasePipe, LowerCasePipe, CurrencyPipe, DatePipe, JsonPipe, SlicePipe],
10. templateUrl: './app.component.html',
11. styleUrl: './app.component.css'
12. })
13. export class AppComponent {
14. nombre = 'Juan Carlos';
15. saldo = 1000.50;
16. dias = ['domingo', 'lunes', 'martes', 'miércoles', 'jueves', 'viernes', 'sábado'];
17. articulos = [{
18. codigo: 1,
19. descripcion: "papas",
20. precio: 12.33
21. }, {
22. codigo: 2,
23. descripcion: "manzanas",
24. precio: 54
25. }];
26. fechaActual = new Date();
27. }

Debemos importar las clases respectivas que contienen las tuberías:

import { UpperCasePipe, LowerCasePipe, CurrencyPipe, DatePipe, JsonPipe, SlicePipe } from '@angular/common';

Hemos definido 5 propiedades de distintos tipos: string, number, vector con los días, vector de artículos con objetos y un objeto de tipo Date.

El archivo de la vista 'app.component.html' queda con la siguiente sintaxis:

<p>Nombre del cliente:{{ nombre }}</p>

<p>Nombre del cliente en mayúsculas:{{ nombre | uppercase }}</p>

<p>Nombre del cliente en minúsculas:{{ nombre | lowercase }}</p>

<p>El saldo es:{{ saldo | currency:'$'}}</p>

<p>Días laborables: {{ dias | slice:1:6}}</p>

<p>{{ articulos | json }}</p>

<p>Fecha actual:{{ fechaActual | date:'d/M/y' }}</p>

<router-outlet />

Ya vimos las pipes: uppercase y lowercase:

<p>Nombre del cliente en mayúsculas:{{ nombre | uppercase }}</p>

<p>Nombre del cliente en minúsculas:{{ nombre | lowercase }}</p>

Para mostrar un importe monetario podemos utilizar la pipe 'currency':

<p>El saldo es:{{ saldo | currency:'$'}}</p>

Si necesitamos mostrar parte de un vector podemos utilizar la pipe 'slice' e indicar en dos parámetros a partir de que índice recuperar los datos y hasta que índice:

<p>Días laborables: {{ dias | slice:1:6}}</p>

Otra pipe útil cuando tenemos que depurar programas es la impresión del contenido de una variable que almacena datos en formato JSON:

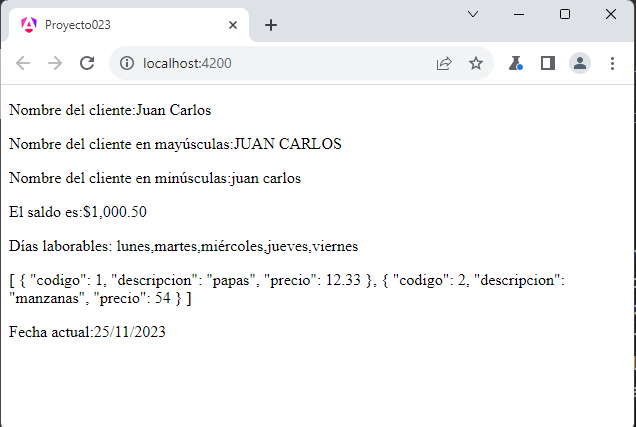
<p>{{ articulos | json }}</p>

Finalmente utilizamos la pipe 'date':

<p>Fecha actual:{{ fechaActual | date:'d/M/y' }}</p>

1. Para ver el resultado en el navegador ejecutamos la aplicación:
2. f:\angulardevya\proyecto023> ng serve -o

Y tenemos como resultado en el navegador:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto023/).

1. Pipes: creación de pipes personalizadas

Vimos en el concepto anterior que una pipe tiene por objetivo convertir un dato en la vista de la componente (html) con la finalidad que el usuario tenga una mejor experiencia.

El framework Angular provee un conjunto de pipes por defecto como pueden ser: uppercase, lowercase, json, date etc

Pero lo más importante que tenemos es que Angular nos permite crear nuestras propias pipes que se adapten a resolver problemas de nuestra aplicación.

Veremos los pasos para crear una pipe haciendo uso de la herramienta Angular CLI para facilitar su codificación.

Problema

Crear una pipe que permita transformar un valor entero comprendido entre 1 y 7 en un string en castellano de dicho número. Además permitir pasar un parámetro opcional indicando que muestre dicho valor en 'ingles' o 'portugues'.

1. Desde la línea de comandos de Node.js procedemos a crear el proyecto024:
2. f:\angulardevya> ng new proyecto024
3. Para crear una 'pipe' descendemos primero a la carpeta del proyecto que acabamos de crear y mediante angular Cli procedemos a ejecutar :
4. f:\angulardevya\proyecto024> ng generate pipe letras

Luego de esto tenemos creados dos archivos:

letras.pipe.ts

letras.pipe.spec.ts

1. El siguiente paso es codificar la clase 'LetrasPipe' que se encuentra almacenada en el archivo 'letras.pipe.ts':
2. import { Pipe, PipeTransform } from '@angular/core';
3. @Pipe({
4. name: 'letras'
5. })
6. export class LetrasPipe implements PipeTransform {
7. transform(value: unknown, ...args: unknown[]): unknown {
8. return null;
9. }
10. }

El método transform es el que se ejecuta cada vez que llamamos a la pipe 'letras' declarada en el decorador @Pipe.

El método transform recibe un parámetro obligatorio que es 'value' y puede ser de cualquier tipo (number, string, array, object etc.) y un segundo parámetro opcional.

Veamos la lógica como implementar la pipe que transforme un valor entero comprendido entre 1 y 7 a letras:

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'letras'

})

export class LetrasPipe implements PipeTransform {

transform(value: any, args?: any): any {

if (args != null) {

if (args=='ingles')

switch (value) {

case 1: return 'one';

case 2: return 'two';

case 3: return 'three';

case 4: return 'four';

case 5: return 'five';

case 6: return 'six';

case 7: return 'seven';

}

if (args=='portugues')

switch (value) {

case 1: return 'um';

case 2: return 'dois';

case 3: return 'três';

case 4: return 'quatro';

case 5: return 'cinco';

case 6: return 'seis';

case 7: return 'sete';

}

}

switch (value) {

case 1: return 'uno';

case 2: return 'dos';

case 3: return 'tres';

case 4: return 'cuatro';

case 5: return 'cinco';

case 6: return 'seis';

case 7: return 'siete';

}

return null;

}

}

El parámetro 'value' recibe el valor que debe procesar la pipe. El parámetro 'args' puede no llegar en dicho caso almacena un null.

Si en 'args' llega un null luego ejecuta el último switch de la función retornando en castellano el número almacenado en 'value'.

Luego en la componente importamos la clase LetrasPipe y definimos por ejemplo un vector con los números del 1 al 7 ('app.component.ts'):

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

import { RouterOutlet } from '@angular/router';

import { LetrasPipe } from './letras.pipe';

@Component({

selector: 'app-root',

imports: [RouterOutlet, LetrasPipe],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

vector = [1,2,3,4,5,6,7];

}

En el archivo 'app.component.html' es donde hacemos uso de la pipe 'letras' que acabamos de crear:

<h1>Números en castellano</h1>

<ul>

@for(valor of vector;track valor) {

<li>

{{valor | letras}}

</li>

}

</ul>

<h1>Números en inglés</h1>

<ul>

@for(valor of vector;track valor) {

<li>

{{valor | letras:'ingles'}}

</li>

}

</ul>

<h1>Números en portugués</h1>

<ul>

@for(valor of vector;track valor) {

<li>

{{valor | letras:'portugues'}}

</li>

}

</ul>

<router-outlet />

La primer forma de llamar a la pipe sin parámetro es:

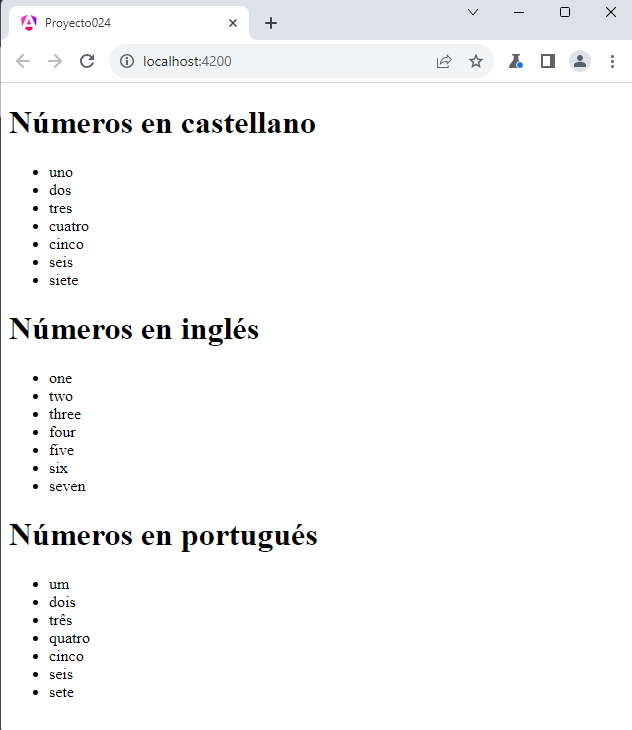
{{valor | letras}}

De esta forma tenemos en pantalla los valores de los números en castellano.

En el caso que le pasemos como parámetro alguno de los dos valores 'ingles' o 'portugues' tendremos como resultado la transformación del número a alguno de estos dos idiomas:

{{valor | letras:'ingles'}}

1. Y tenemos como resultado en el navegador:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto024/).

1. Compilación y despliegue de una aplicación Angular en un servidor de Internet

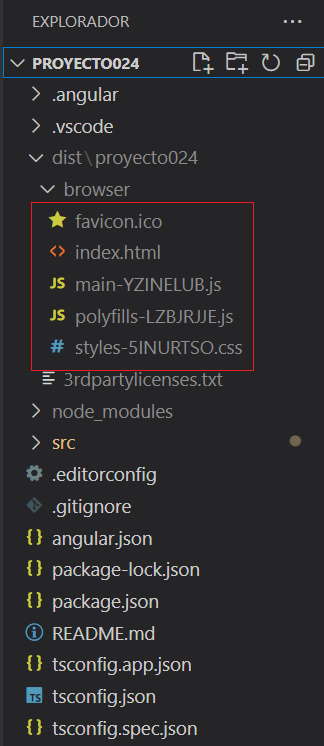
Cuando uno desarrolla una aplicación web utilizando el framework de Angular lo desarrolla y prueba en forma local. Cada cambio que desarrolla lo puede probar en forma local ejecutando el servidor de desarrollo que trae Angular:

f:\angulardevya\proyecto024>ng serve -o

Una vez finalizada la aplicación debemos subirla a nuestro servidor real en Internet, para esto debemos ejecutar el siguiente comando de Angular CLI:

f:\angulardevya\proyecto024>ng build

Luego de este proceso se genera una carpeta llamada 'dist' que contiene todos los archivos que debemos subir a nuestro servidor de Internet:



Los archivos de esta carpeta se deben subir a la carpeta raíz de nuestro servidor de Internet, luego cuando accedemos al dominio de nuestro sitio:

www.misitio.com

El servidor responde devolviendo el archivo 'index.html' y este en su interior tiene todas las referencias a archivos \*.js y \*.css

<!doctype html>

<html lang="en" data-critters-container>

<head>

<meta charset="utf-8">

<title>Proyecto024</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

<link rel="stylesheet" href="styles-5INURTSO.css"></head>

<body>

<app-root></app-root>

<script src="polyfills-LZBJRJJE.js" type="module"></script><script src="main-YZINELUB.js" type="module"></script></body>

</html>

Subir una aplicación Angular a una subcarpeta de nuestro servidor.

Si nuestra aplicación Angular no se ejecutará en la raíz de nuestro servidor de Internet, el proceso de compilación es diferente.

Por ejemplo si queremos cargar una aplicación Angular a la carpeta:

http://ejerciciostutorialesya.com/angulardev/proyecto024/

El proceso de compilación debe ser:

f:\angulardevya\proyecto024>ng build --base-href=/angulardev/proyecto024/

Una vez construido el proyecto debemos subir todos los archivos a la carpeta /angulardev/proyecto024/ y ya podemos probar la aplicación: [aquí](http://ejerciciostutorialesya.com/angulardev/proyecto024/)

1. Herramienta Angular CLI

Una herramienta fundamental cuando desarrollamos proyectos es la aplicación Angular CLI, que la hemos utilizado desde el primer día.

Hemos visto que Angular CLI nos permite entre otras cosas:

* Crear una aplicación de Angular (ng new proyecto001)
* Ejecutar una aplicación Angular en forma local (ng serve -o)
* Crear componentes (ng generate component dado)
* Crear servicios (ng generate service articulos)
* Crear tuberías (ng generate pipe letras)
* Desplegar aplicaciones en producción (ng build)

Debe tener siempre a mano la página que contiene la documentación oficial de [Angular CLI](https://angular.dev/tools/cli).

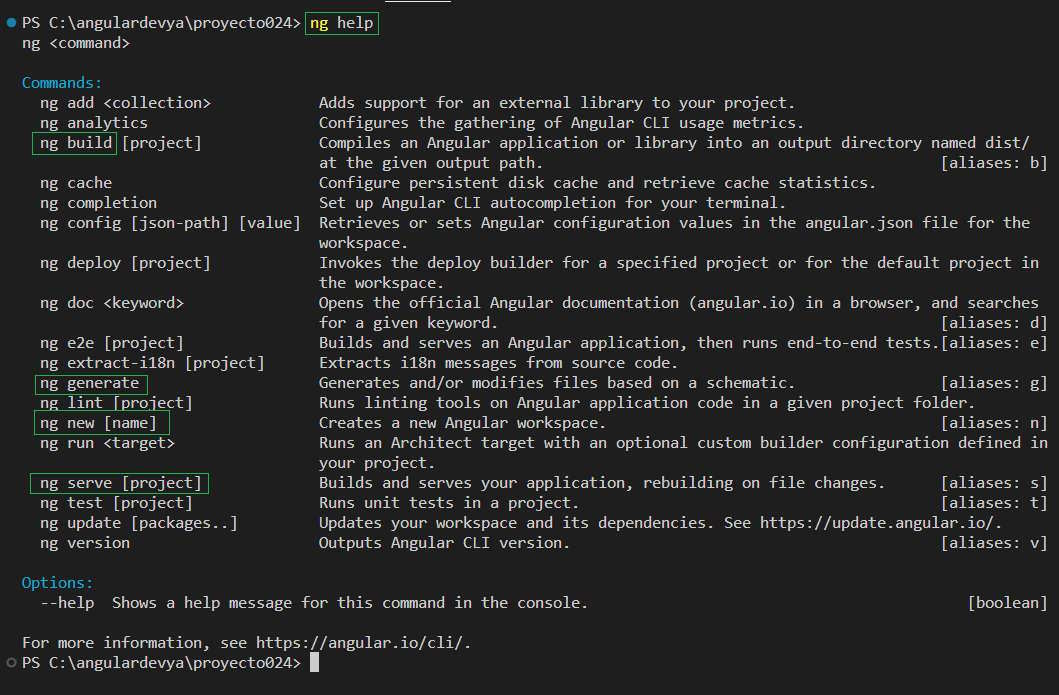
Recordemos que en el primer concepto de este curso lo primero que hicimos luego de instalar Node.js es la instalación de Angular.CLI mediante el comando:

npm install -g @angular/cli

Comandos útiles de Angular.CLI

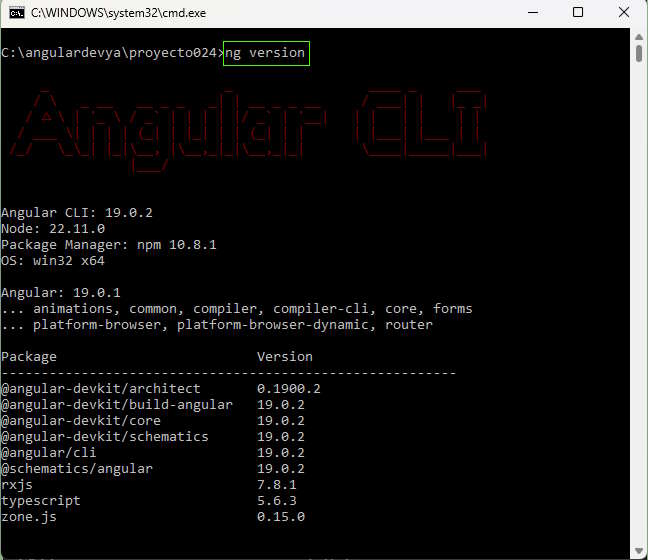
* Para conocer todos los comandos disponibles de la herramienta Angular.CLI debemos ejecutar:
* ng help

Tenemos como resultado muchos de los comandos ya vistos y utilizados en conceptos anteriores:



Si por ejemplo queremos ver la versión de Angular.CLI que tenemos instalada en nuestra computadora debemos ejecutar:

ng version

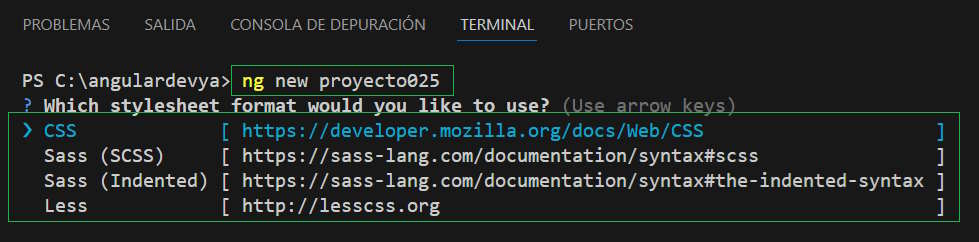


1. Herramienta Angular CLI - comando: ng new

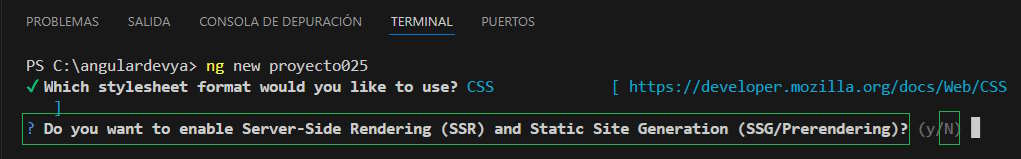
Hemos visto que con el comando 'new' de la herramienta Angular CLI se crea un esqueleto de una aplicación Angular:

ng new proyecto025

Durante la generación del proyecto se nos consulta como queremos administrar las hojas de estilo (por defecto archivos CSS):



Seguidamente se nos cuestiona si queremos activar el renderizado en el servidor (por defecto si presionamos la tecla enter se selecciona que NO queremos):



El comando new podemos pasar otros parámetros para adaptar la aplicación generada.

* Podemos especificar el prefijo a los selectores generados mediante el parámetro --prefix:
* ng new proyecto025 --prefix ele

Luego se crea con prefijo 'ele' en lugar del valor por defecto 'app':

Si abrimos la componente creada por defecto podemos ver que el prefijo es 'ele':

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'ele-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

title = 'proyecto025';

}

En lugar de 'app':

selector: **'app-root'**

Luego si creamos una componente dentro del proyecto también se respetará el prefijo definido en la creación del proyecto:

ng generate component dado

El selector de la componente dado queda con el siguiente nombre:

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

@Component({

selector: 'ele-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

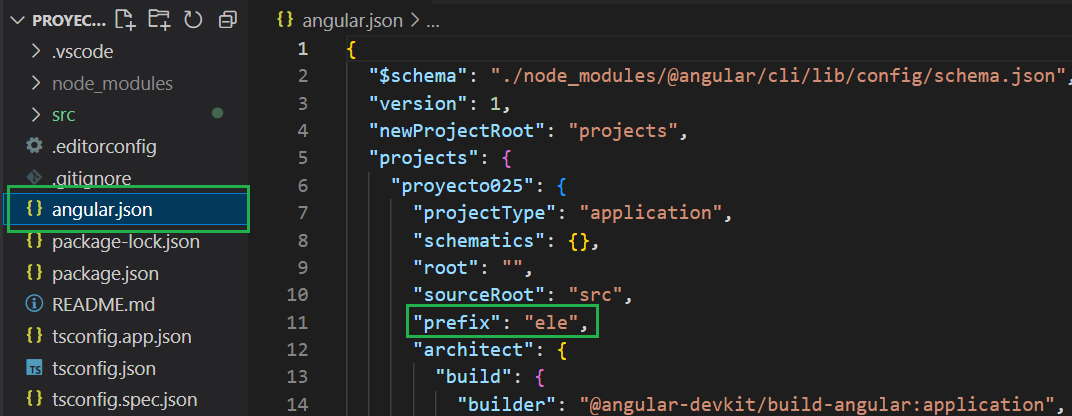
export class DadoComponent {

}

La definición de prefijos en las componentes de Angular permiten diferenciar las etiquetas nativas del navegador (ej. 'button') con las etiquetas propias de Angular (ej. ele-button)

Al crearse un proyecto uno de los archivos fundamentales que almacenan datos de configuración del mismo es 'angular.json' (se encuentra en la carpeta raíz del proyecto)

En el archivo 'angular.json' se encuentra almacenado el prefijo definido al crearse el proyecto:

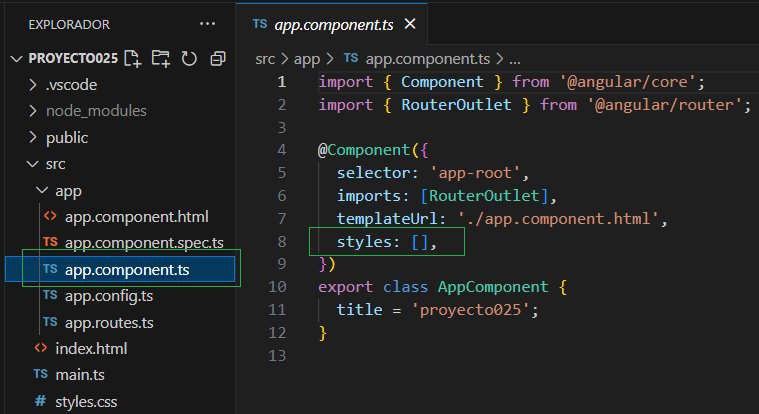


Si queremos escribir menos con Angular CLI al crear el proyecto podemos utilizar el alias -p en lugar de --prefix:

ng new proyecto025 -p ele

* Mediante el parámetro --inline-style (alias: -s) podemos evitar que se cree el archivo \*.css y la definición de los estilos se deba hacer directamente en el archivo \*.ts:
* ng new proyecto025 --inline-style

Con esto no se crearán los archivos \*.css para las componentes. El archivo \*.ts de la componente tienen un lugar donde definir los estilos:



Puede tener sentido si la componente es bastante sencilla y no requiere la definición de una hoja de estilo compleja.

Probar modificar los archivos \*.ts y \*.html de la componente del proyecto.

app.component.ts

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styles: ['.titulo { color:red; font-size:2rem }',

'.parrafo {color:black; font-size:1.1rem }'],

})

export class AppComponent {

tit = 'Prueba de inline-style';

}

app.component.html

<div style="text-align:center">

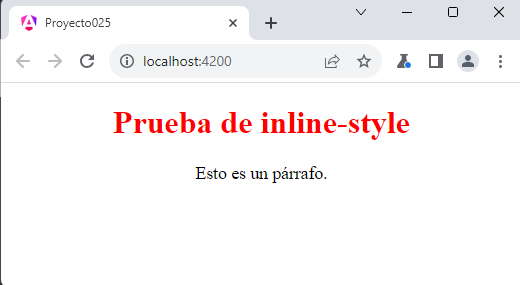
<h1 class="titulo">{{tit}}</h1>

<p class="parrafo">Esto es un párrafo.</p>

</div>

<router-outlet />

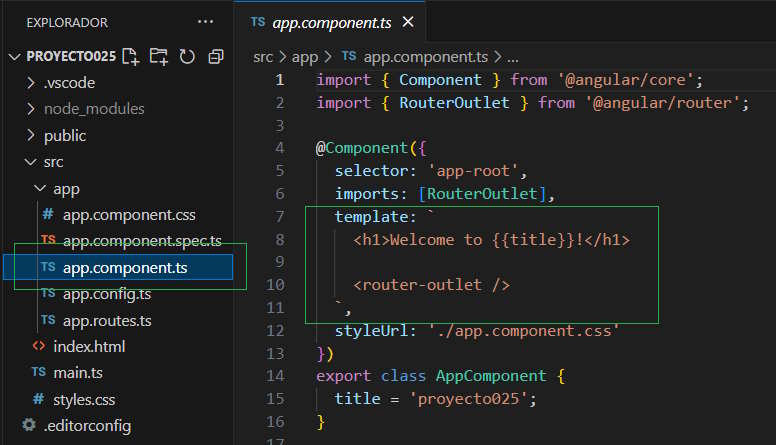
Si ejecutamos el proyecto podemos ver que los estilos se están recuperando del archivo \*.ts:



* Mediante el parámetro --inline-template (alias: -s) podemos evitar que se cree el archivo \*.html y la definición del HTML se hace directamente en el archivo \*.ts (borre primero el proyecto025 antes de crearlo nuevamente con esta nueva configuración):
* ng new proyecto025 --inline-template

Con este parámetro estamos indicando que no se cree el archivo \*.html para la componente y la definición del HTML se haga en el mismo archivo \*.js

Luego de creado el proyecto solo se han creado los archivos \*.css y \*.ts:



Nuevamente decimos que esto tiene sentido si la complejidad de la componente Angular no es grande.

Podemos inclusive evitar que se creen tanto el archivo \*.css y \*.html indicando ambos parámetros al crear la componente:

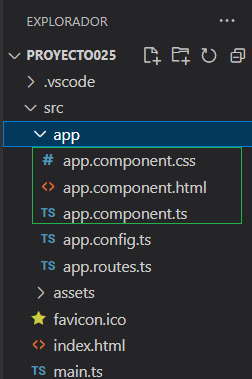
ng new proyecto025 --inline-style --inline-template

* Si tratamos de crear un proyecto y ya existe uno en la carpeta actual se produce un error al ejecutar:
* ng new proyecto025

Si queremos forzar la creación del proyecto y sobreescribir los archivos actuales debemos añadir el parámetro --force (alias -f):

ng new proyecto025 --force

* Si por algún motivo no queremos que Angular.CLI nos genere el archivo de test debemos pasar el parámetro --skip-tests (alias -s):
* ng new proyecto025 --skip-tests

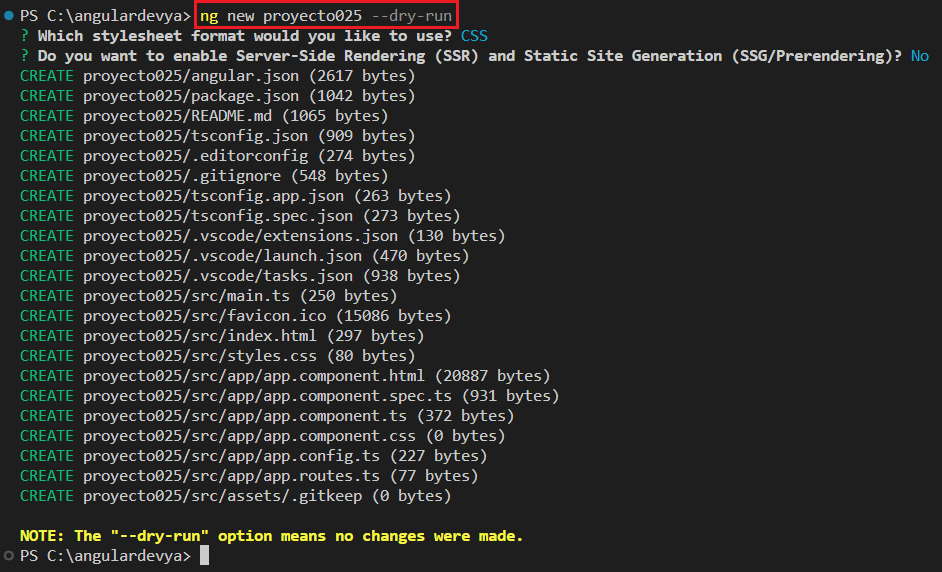


Como podemos comprobar no se ha generado el archivo 'app.component.spec.ts'.

* Otro parámetro más que podemos utilizar cuando vamos a crear un proyecto es --dry-run (alias: -d)

Mediante esta opción Angular CLI nos informa que archivos se crearán (no los crea) y a partir de estos datos poder tomar la decisión de crear o no el proyecto:

ng new proyecto025 --dry-run



1. Herramienta Angular CLI - comando: ng generate

Otro comando que hemos utilizado a lo largo de los primeros conceptos de este curso de Angular es 'generate'.

Mediante el comando 'generate' de la herramienta Angular CLI podemos crear:

* component
* service
* module
* pipe
* class
* interface
* enum
* directive
* application
* library
* guard
* service-worker
* web-worker
* resolver
* interceptor
* app-shell

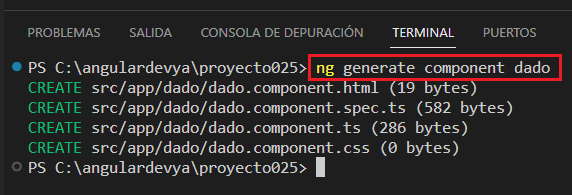
ng generate component

Mediante el comando:

ng generate component dado

Se crea una componente 'dado' con los archivos respectivos.

Se informa en la consola los archivos creados:



Podemos pasar varias opciones cuando creamos la componente:

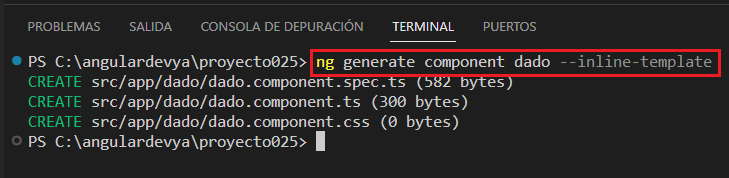
ng generate component dado --inline-style

Evita que se cree el archivo dado.component.css

Lo mismo podemos evitar que se cree el archivo HTML:

ng generate component dado --inline-template

Tenemos como resultado la no creación del archivo HTML:



Podemos definir el prefijo para la componente mediante la opción --prefix (alias: -p):

ng generate component dado --prefix juego

Luego la componente que se crea tiene dicho prefijo:

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

@Component({

selector: 'juego-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

export class DadoComponent {

}

Si queremos definir el nombre completo para el selector tenemos la opción --selector:

ng generate component dado --selector ju-dado

Luego la componente se crea con dicho nombre de selector:

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

@Component({

selector: **'ju-dado'**,

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

export class DadoComponent {

}

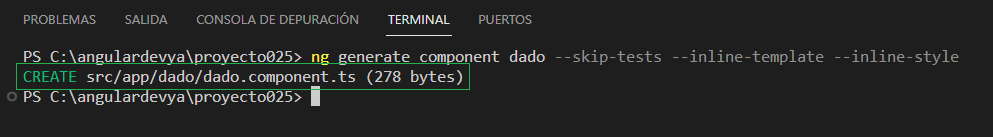
Para evitar que se cree el archivo 'dado.spec.js' debemos insertar el comando --skip-tests:

ng generate component dado --skip-tests

Luego no se genera el archivo 'dado.spec.js'.

Tengamos en cuenta que todas estas opciones se pueden combinar y ejecutar en forma simultanea, por ejemplo si queremos generar solo el archivo \*.ts de la componente y que no genere el archivo spec, \*.css y \*.html:

ng generate component dado --skip-tests --inline-template --inline-style



Otras opciones posibles cuando creamos una componente son:

* --force (alias: -f) Forzar la sobrescritura de los archivos existentes (se borra la componente anterior que tiene el mismo nombre)
* --dry-run (alias: -d) Informa los archivos que se crearán sin hacer dicha actividad.

Angular CLI permite ingresar comandos en formato resumido utilizando el primer caracter:

ng g c dado

En lugar de escribir:

ng generate component dado

ng generate service

Mediante el comando:

ng generate service articulos

Se crea una clase ArticulosService y se inyecta a nivel de 'root':

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

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

constructor() { }

}

Disponemos de los siguientes opciones en este comando:

* --dry-run
* --force
* --skip-tests

ng generate pipe

Mediante el comando:

ng generate pipe letras

Se genera el archivo 'letras.pipe.ts':

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'letras'

})

export class LetrasPipe implements PipeTransform {

transform(value: unknown, ...args: unknown[]): unknown {

return null;

}

}

Disponemos de los siguientes opciones en este comando:

* --dry-run
* --force
* --skip-tests

ng generate class

Mediante el comando:

ng generate class articulo

Se crea el archivo 'articulo.ts':

export class Articulo {

}

ng generate interface

Mediante el comando:

ng generate interface venta

Se crea el archivo 'venta.ts':

export interface Venta {

}

ng generate enum

Mediante el comando:

ng generate enum operaciones

Se crea el archivo 'operaciones.enum.ts':

export enum Operaciones {

}

1. Herramienta Angular CLI - comando: ng serve

Otro comando que hemos utilizado en cada uno de los proyectos que hemos implementado es 'serve' (desde el principio hemos utilizado la opción -o):

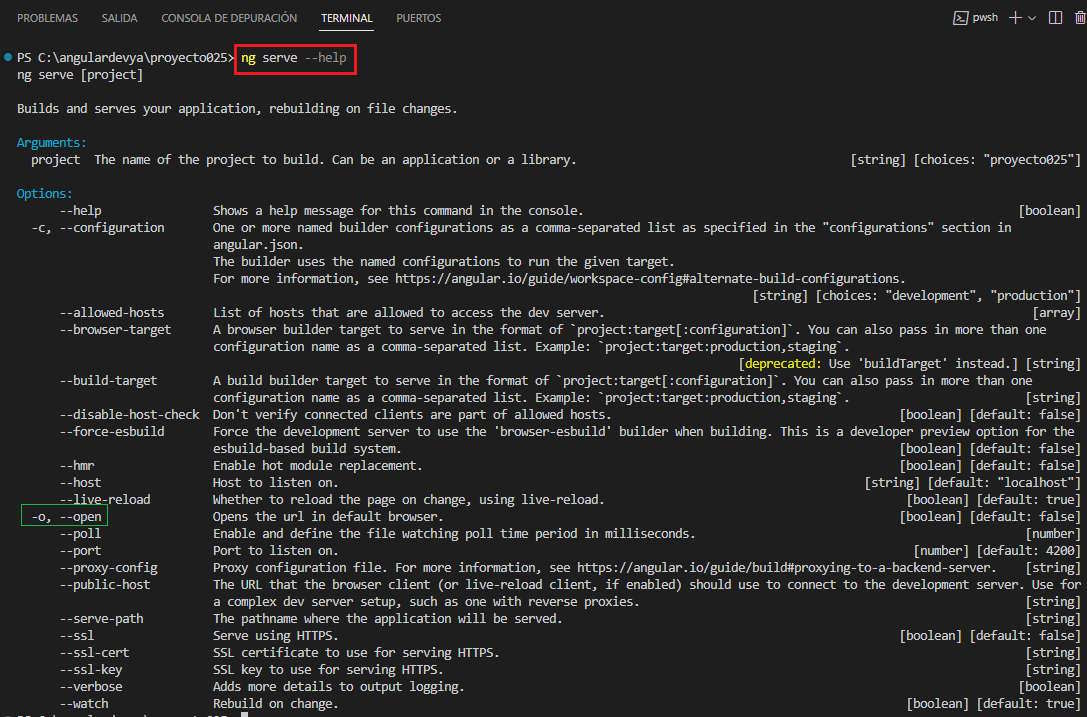
ng serve -o

Recordemos que debemos ejecutar el comando serve en la carpeta donde se haya nuestra aplicación Angular. Al disponer la opción -o se abre automáticamente el navegador web. La sintaxis larga pero que produce la misma acción es:

ng serve --open

Para conocer todas las opciones disponibles en un comando de Angular CLI debemos acceder a la opción --help:

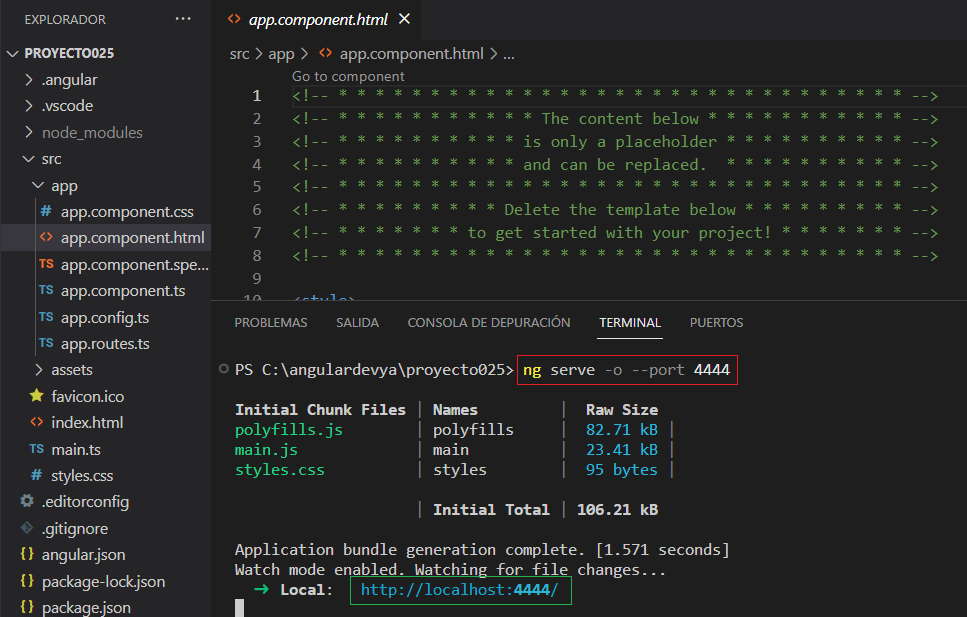
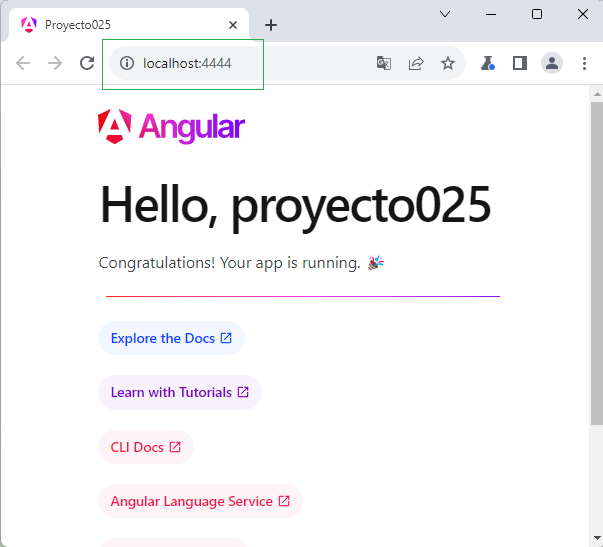
ng serve --help



Opción --port

Por defecto el servidor web local que crea Angular CLI se ejecuta en el puerto 4200, si necesitamos que el servidor web se ejecute en otro puerto podemos indicarlo con la opción --port en el momento de iniciarlo:

ng serve -o --port 4444

Opción --watch (alias: -w)

Por defecto cada vez que modificamos nuestro proyecto y grabamos los cambios el resultado se actualiza en el navegador en forma automática. En algunas situaciones si queremos que no se actualice debemos utilizar la opción 'watch' pasando el valor false:

ng serve -o --watch false

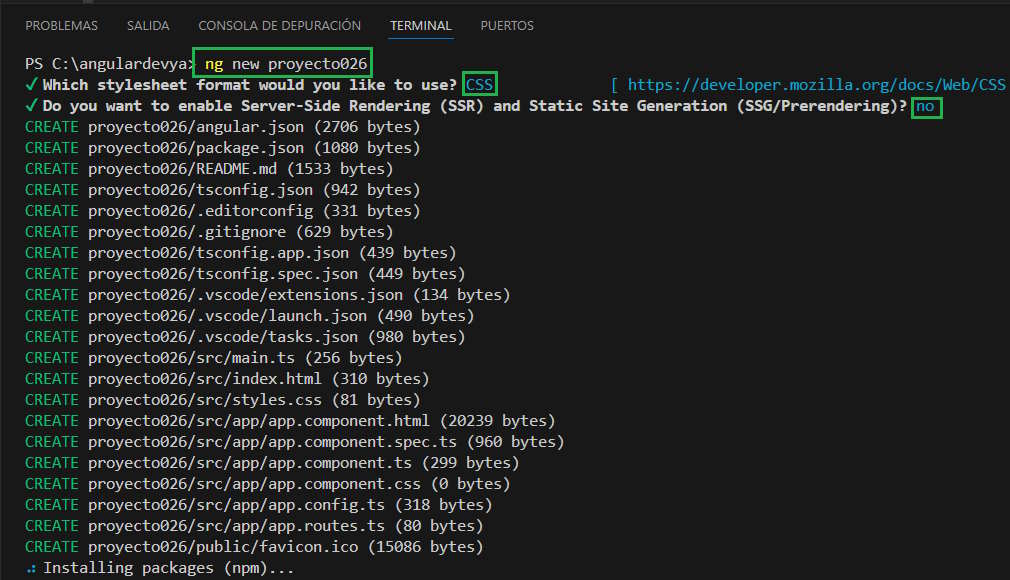
1. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta raiz

Hemos trabajado por un tiempo desarrollando ejercicios con el framework de Angular 19, pero no nos hemos detenido para analizar en profundidad los archivos y carpetas que se generan cada vez que creamos un proyecto.

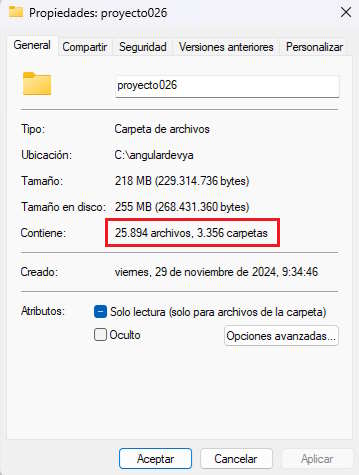
Pana analizar los archivos y carpetas crearemos un nuevo proyecto:

ng new proyecto026

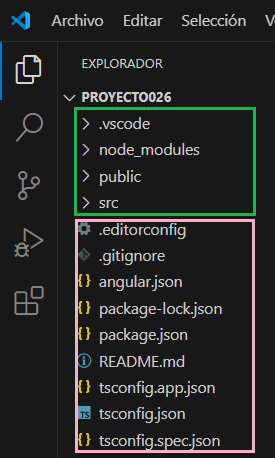
Lo primero que puede variar esta estructura son que decisiones tomamos para indicar si queremos utilizar formato de hojas de estilo y Server-Side rendering (seleccionemos los datos por defecto es decir CSS y no Server-Side rendering):



Si analizamos el contenido de la carpeta que se crea 'proyecto026' nos puede intimidar la cantidad de carpetas y archivos que contiene:



La carpeta 'proyecto026' creada con Angular 19 tiene 9 archivos y 4 carpetas:



Archivos de la raiz del proyecto

* **.editorconfig** Este archivo define estilos de codificación consistentes para múltiples desarrolladores que trabajan en el mismo proyecto con editores de texto distinto. No hay problemas de estilo si un desarrollador utiliza el VS Code y otro el Sublime Text, ya que ambos editores tomarán en cuenta las definiciones contenidas en .editorconfig:
* # Editor configuration, see https://editorconfig.org
* root = true
* [\*]
* charset = utf-8
* indent\_style = space
* indent\_size = 2
* insert\_final\_newline = true
* trim\_trailing\_whitespace = true
* [\*.ts]
* quote\_type = single
* [\*.md]
* max\_line\_length = off
* trim\_trailing\_whitespace = false

Para modificar este archivo podemos consultar el sitio que define su estándar [editorconfig.org](https://editorconfig.org/)

* **.gitignore** Especifica los archivo y carpetas que debe ignorar el sistema de control de versiones (Git)
* # See http://help.github.com/ignore-files/ for more about ignoring files.
* # Compiled output
* /dist
* /tmp
* /out-tsc
* /bazel-out
* # Node
* /node\_modules
* npm-debug.log
* yarn-error.log
* # IDEs and editors
* .idea/
* .project
* .classpath
* .c9/
* \*.launch
* .settings/
* \*.sublime-workspace
* # Visual Studio Code
* .vscode/\*
* !.vscode/settings.json
* !.vscode/tasks.json
* !.vscode/launch.json
* !.vscode/extensions.json
* .history/\*
* # Miscellaneous
* /.angular/cache
* .sass-cache/
* /connect.lock
* /coverage
* /libpeerconnection.log
* testem.log
* /typings
* # System files
* .DS\_Store
* Thumbs.db
* **angular.json** Se definen valores por defecto para la configuración del proyecto para las herramientas de compilación y desarrollo:
* {
* "$schema": "./node\_modules/@angular/cli/lib/config/schema.json",
* "version": 1,
* "newProjectRoot": "projects",
* "projects": {
* "proyecto026": {
* "projectType": "application",
* "schematics": {},
* "root": "",
* "sourceRoot": "src",
* "prefix": "app",
* "architect": {
* "build": {
* "builder": "@angular-devkit/build-angular:application",
* "options": {
* "outputPath": "dist/proyecto026",
* "index": "src/index.html",
* "browser": "src/main.ts",
* "polyfills": [
* "zone.js"
* ],
* "tsConfig": "tsconfig.app.json",
* "assets": [
* {
* "glob": "\*\*/\*",
* "input": "public"
* }
* ],
* "styles": [
* "src/styles.css"
* ],
* "scripts": []
* },
* "configurations": {
* "production": {
* "budgets": [
* {
* "type": "initial",
* "maximumWarning": "500kB",
* "maximumError": "1MB"
* },
* {
* "type": "anyComponentStyle",
* "maximumWarning": "4kB",
* "maximumError": "8kB"
* }
* ],
* "outputHashing": "all"
* },
* "development": {
* "optimization": false,
* "extractLicenses": false,
* "sourceMap": true
* }
* },
* "defaultConfiguration": "production"
* },
* "serve": {
* "builder": "@angular-devkit/build-angular:dev-server",
* "configurations": {
* "production": {
* "buildTarget": "proyecto026:build:production"
* },
* "development": {
* "buildTarget": "proyecto026:build:development"
* }
* },
* "defaultConfiguration": "development"
* },
* "extract-i18n": {
* "builder": "@angular-devkit/build-angular:extract-i18n"
* },
* "test": {
* "builder": "@angular-devkit/build-angular:karma",
* "options": {
* "polyfills": [
* "zone.js",
* "zone.js/testing"
* ],
* "tsConfig": "tsconfig.spec.json",
* "assets": [
* {
* "glob": "\*\*/\*",
* "input": "public"
* }
* ],
* "styles": [
* "src/styles.css"
* ],
* "scripts": []
* }
* }
* }
* }
* }
* }
* "prefix": "app"

"prefix" es una cadena que Angular antepone a los selectores generados. Por ejemplo la componente por defecto que se crea:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrl: './app.component.css'

})

export class AppComponent {

title = 'proyecto026';

}

Si creamos una componente llamada 'dado' luego su selector toma el nombre 'app-dado':

@Component({

selector: 'app-dado',

imports: [],

templateUrl: './dado.component.html',

styleUrl: './dado.component.css'

})

export class DadoComponent {

}

La carpeta raíz para los archivos fuente del proyecto esta definido en:

"sourceRoot": "src"

En la entrada "assets" hacemos referencia a los archivos estáticos del proyecto y las carpetas donde se almacenan, recordemos en el problema donde necesitabamos insertar archivos \*.jpg los cargamos en la carpeta 'assets':

"assets": [

"src/favicon.ico",

"src/assets"

],

En la entrada styles especificamos todos los archivos de hoja de estilo globales al proyecto (styles.css es uno que se ha creado por defecto):

"styles": [

"src/styles.css"

],

Por ejemplo si queremos tener acceso solo a la hoja de estilo de BootStrap 'bootstrap.min.css', solo debemos copiar dicho archivo en la carpeta 'src' o una subcarpeta de la misma y especificar su path:

"styles": [

"src/styles.css",

"src/bootstrap.min.css"

],

En la entrada scripts especificamos los archivos de script JavaScript para agregar al contexto global del proyecto:

"scripts": []

Por ejemplo si queremos tener acceso al archivo JavaScript de BootStrap 'bootstrap.min.js', debemos copiar dicho archivo y el jquery en la carpeta 'src' o una subcarpeta de la misma y especificar su path:

"scripts": ["src/jquery-3.4.1.min.js",

"src/bootstrap.min.js"]

El orden en que los listamos es importante, primero debe ser el jquery y seguidamente el bootstrap.min.js (según sus dependencias, en este caso BootStrap depende de la librería JQuery)

* **package.json** Configura las dependencias del paquete npm que están disponibles para todos los proyectos en el espacio de trabajo.
* {
* "name": "proyecto026",
* "version": "0.0.0",
* "scripts": {
* "ng": "ng",
* "start": "ng serve",
* "build": "ng build",
* "watch": "ng build --watch --configuration development",
* "test": "ng test"
* },
* "private": true,
* "dependencies": {
* "@angular/animations": "^19.0.0",
* "@angular/common": "^19.0.0",
* "@angular/compiler": "^19.0.0",
* "@angular/core": "^19.0.0",
* "@angular/forms": "^19.0.0",
* "@angular/platform-browser": "^19.0.0",
* "@angular/platform-browser-dynamic": "^19.0.0",
* "@angular/router": "^19.0.0",
* "rxjs": "~7.8.0",
* "tslib": "^2.3.0",
* "zone.js": "~0.15.0"
* },
* "devDependencies": {
* "@angular-devkit/build-angular": "^19.0.2",
* "@angular/cli": "^19.0.2",
* "@angular/compiler-cli": "^19.0.0",
* "@types/jasmine": "~5.1.0",
* "jasmine-core": "~5.4.0",
* "karma": "~6.4.0",
* "karma-chrome-launcher": "~3.2.0",
* "karma-coverage": "~2.2.0",
* "karma-jasmine": "~5.1.0",
* "karma-jasmine-html-reporter": "~2.1.0",
* "typescript": "~5.6.2"
* }
* }

El archivo se organiza indicando primero en la propiedad "dependencies" todos los paquetes requeridos para la aplicación que estamos desarrollando, y en segundo lugar en la propiedad "devDepencencies" se hace referencia a todos los paquetes requeridos para desarrollar la aplicación.

Los paquetes que se declaran en el archivo corresponden a:

Paquetes de Angular: núcleo angular y módulos opcionales; comienzan los nombres de sus paquetes @angular/

Paquetes de soporte : bibliotecas de terceros que deben estar presentes para que se ejecuten las aplicaciones de Angular

Paquetes Polyfill : Polyfills tapan huecos en la implementación de JavaScript que no disponen algunos navegadores.

* **package-lock.json**Proporciona información de versión para todos los paquetes instalados en la carpeta node\_modules por el cliente npm (no lo mostramos ya que tiene más de 15000 líneas)
* **tsconfig.json** Especifica la configuración predeterminada del lenguaje TypeScript para proyectos en el espacio de trabajo.
* /\* To learn more about Typescript configuration file: https://www.typescriptlang.org/docs/handbook/tsconfig-json.html. \*/
* /\* To learn more about Angular compiler options: https://angular.dev/reference/configs/angular-compiler-options. \*/
* {
* "compileOnSave": false,
* "compilerOptions": {
* "outDir": "./dist/out-tsc",
* "strict": true,
* "noImplicitOverride": true,
* "noPropertyAccessFromIndexSignature": true,
* "noImplicitReturns": true,
* "noFallthroughCasesInSwitch": true,
* "skipLibCheck": true,
* "isolatedModules": true,
* "esModuleInterop": true,
* "experimentalDecorators": true,
* "moduleResolution": "bundler",
* "importHelpers": true,
* "target": "ES2022",
* "module": "ES2022"
* },
* "angularCompilerOptions": {
* "enableI18nLegacyMessageIdFormat": false,
* "strictInjectionParameters": true,
* "strictInputAccessModifiers": true,
* "strictTemplates": true
* }
* }
* **tsconfig.spec.json** Configuración de TypeScript para las pruebas de aplicación.
* /\* To learn more about Typescript configuration file: https://www.typescriptlang.org/docs/handbook/tsconfig-json.html. \*/
* /\* To learn more about Angular compiler options: https://angular.dev/reference/configs/angular-compiler-options. \*/
* {
* "extends": "./tsconfig.json",
* "compilerOptions": {
* "outDir": "./out-tsc/spec",
* "types": [
* "jasmine"
* ]
* },
* "include": [
* "src/\*\*/\*.spec.ts",
* "src/\*\*/\*.d.ts"
* ]
* }
* **tsconfig.app.json**Configuración de TypeScript específica de la aplicación , incluidas las opciones de compilación de plantillas de TypeScript y Angular.
* /\* To learn more about Typescript configuration file: https://www.typescriptlang.org/docs/handbook/tsconfig-json.html. \*/
* /\* To learn more about Angular compiler options: https://angular.dev/reference/configs/angular-compiler-options. \*/
* {
* "extends": "./tsconfig.json",
* "compilerOptions": {
* "outDir": "./out-tsc/app",
* "types": []
* },
* "files": [
* "src/main.ts"
* ],
* "include": [
* "src/\*\*/\*.d.ts"
* ]

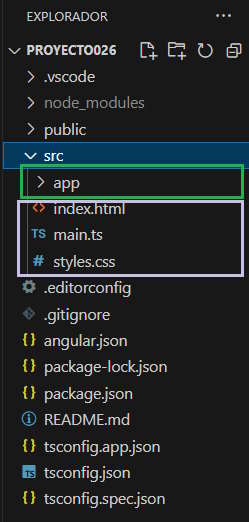
}

* **README.md**Primer archivo que deberíamos leer cuando accedemos a un proyecto nuevo.
* # Proyecto026
* This project was generated using [Angular CLI](https://github.com/angular/angular-cli) version 19.0.2.
* ## Development server
* To start a local development server, run:
* ```bash
* ng serve
* ```
* Once the server is running, open your browser and navigate to `http://localhost:4200/`. The application will automatically reload whenever you modify any of the source files.
* ## Code scaffolding
* Angular CLI includes powerful code scaffolding tools. To generate a new component, run:
* ```bash
* ng generate component component-name
* ```
* For a complete list of available schematics (such as `components`, `directives`, or `pipes`), run:
* ```bash
* ng generate --help
* ```
* ## Building
* To build the project run:
* ```bash
* ng build
* ```
* This will compile your project and store the build artifacts in the `dist/` directory. By default, the production build optimizes your application for performance and speed.
* ## Running unit tests
* To execute unit tests with the [Karma](https://karma-runner.github.io) test runner, use the following command:
* ```bash
* ng test
* ```
* ## Running end-to-end tests
* For end-to-end (e2e) testing, run:
* ```bash
* ng e2e
* ```
* Angular CLI does not come with an end-to-end testing framework by default. You can choose one that suits your needs.
* ## Additional Resources

For more information on using the Angular CLI, including detailed command references, visit the [Angular CLI Overview and Command Reference](https://angular.dev/tools/cli) page.

1. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta 'src'

La carpeta 'src' (**s**ou**rc**e) depende directamente de la carpeta raiz del proyecto y contiene en su interior 1 carpetas y 3 archivos:



Archivos de la carpeta 'src'

* **index.html** Es la página HTML principal que se sirve cuando alguien visita el sitio web. Angular CLI agrega automáticamente todos los archivos JavaScript y CSS al compilar la aplicación, por lo que generalmente no necesita agregar ningún <script>o <link>.
* <!doctype html>
* <html lang="en">
* <head>
* <meta charset="utf-8">
* <title>Proyecto026</title>
* <base href="/">
* <meta name="viewport" content="width=device-width, initial-scale=1">
* <link rel="icon" type="image/x-icon" href="favicon.ico">
* </head>
* <body>
* <app-root></app-root>
* </body>
* </html>

Si ejecutamos la aplicación y vemos el código que llega al navegador nos encontraremos un contenido similar a este (se agregan una serie de archivos \*.js):

<!doctype html>

<html lang="en">

<head>

<script type="module" src="/@vite/client"></script>

<meta charset="utf-8">

<title>Proyecto026</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

<link rel="stylesheet" href="styles.css"></head>

<body>

<app-root></app-root>

<script src="polyfills.js" type="module"></script><script src="main.js" type="module"></script></body>

</html>

Podemos analizar que en el archivo original en el body se inserta una etiqueta de tipo 'app-root'. Esta etiqueta hace referencia a la única componente que se ha creado en forma automática y se encuentra almacenada en la carpeta hija llamada 'app' donde si abrimos el archivo 'app.component.ts' nos encontramos definido en la propiedad 'selector' el nombre de la etiqueta 'app-root':

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: **'app-root'**,

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'proyecto026';

}

Si borraramos de la sección del body del archivo 'index.html' la etiqueta 'app-root' y lanzamos la aplicación, veremos que aparece una página web vacía.

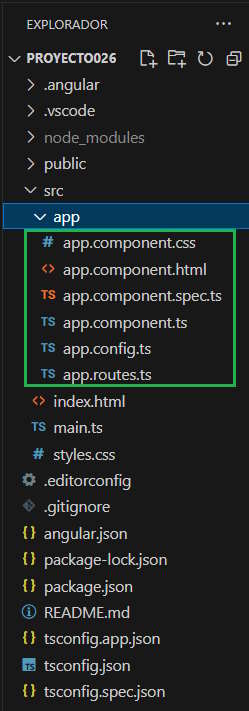
* **styles.css** En este archivo definimos la hoja de estilo global del proyecto.
* **main.ts** Este archivo es el punto de entrada principal para la aplicación Angular:
* import { bootstrapApplication } from '@angular/platform-browser';
* import { appConfig } from './app/app.config';
* import { AppComponent } from './app/app.component';
* bootstrapApplication(AppComponent, appConfig)
* .catch((err) => console.error(err));

Se llama a la función bootstrapApplication y le pasamos en el primer parámetro la referencia de la componente principal.

1. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta 'app'

Después de haber trabajado con muchos ejercicios podemos tener una idea de como se organizan las carpetas y archivos en un proyecto Angular.

La carpeta 'app' depende directamente de la carpeta 'src' y contiene en su interior 6 archivos:



Archivos de la carpeta 'app'

* **app.component.ts** Define la lógica para el componente raíz de la aplicación, llamado AppComponent:
* import { Component } from '@angular/core';
* import { RouterOutlet } from '@angular/router';
* @Component({
* selector: 'app-root',
* imports: [RouterOutlet],
* templateUrl: './app.component.html',
* styleUrl: './app.component.css'
* })
* export class AppComponent {
* title = 'proyecto026';
* }

Hay que tener en cuenta que la clase se llama 'AppComponent' y tiene asociado el selector 'app-root' que es el que hacemos referencia en el archivo index.html.

Angular CLI crea en forma automática esta clase y define en su interior un atributo llamado 'title' y se le asigna un string que coincide con el nombre del proyecto que creamos con el comando 'ng new proyecto037'

* **app.component.html** Define la plantilla HTML asociada con la clase AppComponent. El contenido original tiene más de 300 líneas y seguramente lo eliminaremos para crear la interfaz visual de la componente raiz de nuestra aplicación.

Por ejemplo podemos modificar y disponer que muestre el nombre del proyecto dentro de una etiqueta HTML h1:

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

<router-outlet />

* **app.component.css** Define la hoja de estilo que se aplica solo a dicha componente. Angular CLI crea este archivo vacío.
* **app.component.spec.ts** Se definen las pruebas unitarias para la componente 'AppComponent'
* import { TestBed } from '@angular/core/testing';
* import { AppComponent } from './app.component';
* describe('AppComponent', () => {
* beforeEach(async () => {
* await TestBed.configureTestingModule({
* imports: [AppComponent],
* }).compileComponents();
* });
* it('should create the app', () => {
* const fixture = TestBed.createComponent(AppComponent);
* const app = fixture.componentInstance;
* expect(app).toBeTruthy();
* });
* it(`should have the 'proyecto026' title`, () => {
* const fixture = TestBed.createComponent(AppComponent);
* const app = fixture.componentInstance;
* expect(app.title).toEqual('proyecto026');
* });
* it('should render title', () => {
* const fixture = TestBed.createComponent(AppComponent);
* fixture.detectChanges();
* const compiled = fixture.nativeElement as HTMLElement;
* expect(compiled.querySelector('h1')?.textContent).toContain('Hello, proyecto026');
* });
* });
* **app.routes.ts** El archivo contiene un arreglo de tipo Routes con todas las rutas de nuestra aplicación, por defecto el arreglo se encuentra vacío.
* import { Routes } from '@angular/router';
* export const routes: Routes = [];
* **app.config.ts** Lista de proveedores que deberían estar disponibles para el componente raíz y todos sus hijos.
* import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
* import { provideRouter } from '@angular/router';
* import { routes } from './app.routes';
* export const appConfig: ApplicationConfig = {
* providers: [provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes)]

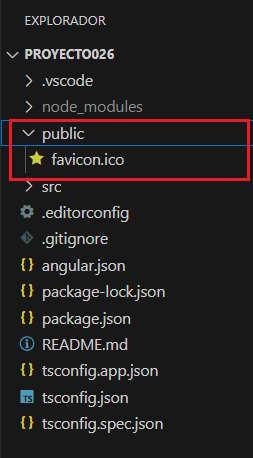
};

1. Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta 'public'

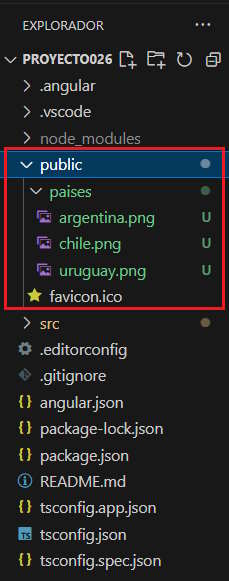
A partir de la versión 18 de Angular se crea por defecto la carpeta public conteniendo el archivo 'favicon.ico'.

En la carpeta public disponemos archivos de imágenes, videos y otros archivos (por ejemplo si tenemos una serie de archivos pdf que se descargan de nuestra aplicación) que se copiarán tal cual en la aplicación definitiva.

La carpeta public en una carpeta hija del proyecto propiamente dicho:



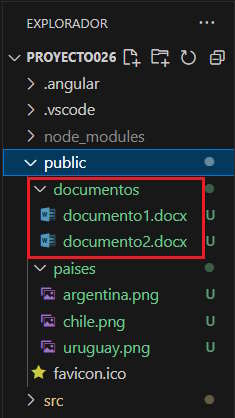
Podemos organizar en esta carpeta, subcarpetas agrupando las imágenes, videos y otros archivos. Crearemos una carpeta llamada 'paises' dentro de la carpeta 'public' e insertamos 3 imágenes de banderas de dichos paises:



Luego accedemos a las mismas indicando el siguiente camino:

<img src="/paises/uruguay.png">

Por ejemplo podríamos disponer una serie de documentos \*.docx para ser descargados desde el sitio:

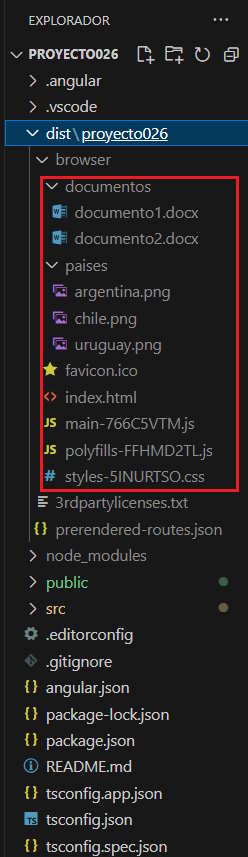


Luego accedemos a los documentos para ser descargados con el siguiente camino:

<a href="/documentos/documento1.docx">Documento 1</a>

Cuando compilamos la aplicación de Angular el contenido de la carpeta 'public' queda sin cambios y debe ser subida al servidor de internet junto con el resto de archivos.

Luego de compilar la aplicación se genera la carpeta 'dist' en la raiz del proyecto, si vemos su contenido nos encontramos que tenemos una copia con todo el contenido de la carpeta 'public':

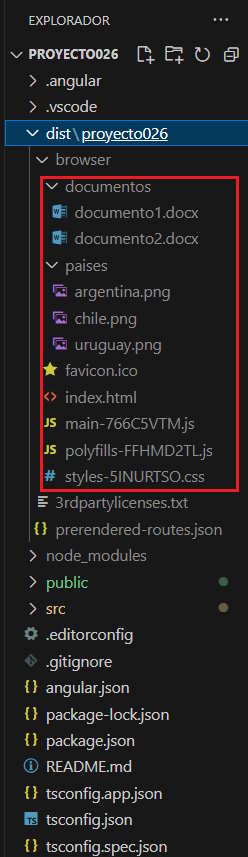


39 - Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpetas '<Nombre del proyecto>\dist'

Las carpetas '<Nombre del proyecto>\dist' no se generan inmediatamente al crear un proyecto Angular, sino cuando compilamos por primera vez:

ng build

Luego de compilar el proyecto Angular por primera vez podemos ver que se han generado las carpetas 'proyecto36\dist' dependiendo de la raiz:



Dentro de la carpeta 'proyecto37' se localizan todos los archivos y carpeta 'assets' que deben ser subidos al servidor web.

* **index.html** El archivo index.html no es exactamente el mismo archivo que se encuentra en la raiz del proyecto, sino con los script necesarios de la aplicación Angular:
* <!doctype html>
* <html lang="en" data-critters-container>
* <head>
* <meta charset="utf-8">
* <title>Proyecto026</title>
* <base href="/angulardev/proyecto026/">
* <meta name="viewport" content="width=device-width, initial-scale=1">
* <link rel="icon" type="image/x-icon" href="favicon.ico">
* <link rel="stylesheet" href="styles-5INURTSO.css"></head>
* <body>
* <app-root></app-root>
* <script src="polyfills-LZBJRJJE.js" type="module"></script><script src="main-BYMF447A.js" type="module"></script></body>
* </html>

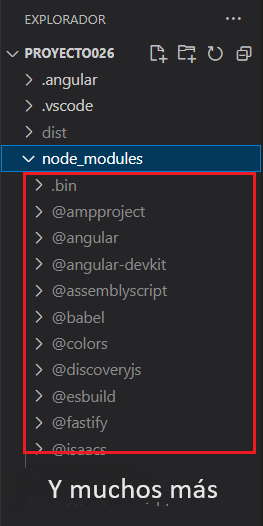
Si analizamos el código añadido podemos ver que se han agregado etiquetas 'script'.

* **styles.xxxxxxxx.css** Contiene una copia de la hoja de estilo global del proyecto (borra saltos de línea y espacios en blanco para reducir su tamaño), recordemos que se almacena en la carpeta 'src' el archivo 'styles.css'.
* **mainxxxxxx.js** Contiene el código JavaScript de la aplicación.
* **polyfillsxxxxxx.js** Contiene más código JavaScript de la aplicación.
* **favicon.ico** Es la copia del archivo que se encuentra en la carpeta raiz del proyecto.
* **3rdpartylicenses.txt** Contiene las licencias de todos los módulos de terceros utilizados en el proyecto:

40 - Estructura y nombres de archivos y carpetas de un proyecto Angular - Carpeta node\_modules

El framework de Angular, el gestor de línea de comandos Angular CLI y todas las componentes que implementamos y utilizamos de otros desarrolladores (por ejemplo Angular Material) son empaquetadas utilizando el software npm (**n**ode **p**ackage **m**anager - es el sistema de gestión de paquetes por defecto para Node.js y maneja las dependencias para una aplicación)

Los paquetes utilizando npm se descargan en la carpeta 'node\_modules':



Por ejemplo en la versión 19 de Angular cuando creamos un proyecto tenemos en principio 555 carpetas que dependen directamente de la carpeta nodo\_modules (y cada carpeta tiene subcarpetas) y a medida que instalemos otros recursos (por ejemplo Angular Material) esta cantidad crecerá.

Hay que tener bien en claro que todos estas carpetas no se instalarán en la aplicación web que desarrollemos, muchas son requeridas por el framework de Angular, otras para las pruebas de integración en Angular, otras para las pruebas unitarias etc.

Esta carpeta si la borramos completamente, la podemos recrear en forma exacta mediante el comando (nos tenemos que posicionar en la carpeta que contiene el archivo package.json):

npm install

Es por ello si tenemos que enviarnos el proyecto por email por ejemplo no es necesario incorporar todos los archivos de esta carpeta, ya que los podemos recrear.

El archivo 'package.json' que se encuentra en la carpeta raiz de nuestro proyecto es el que tiene la información de todos los paquetes requeridos:

{

"name": "proyecto026",

"version": "0.0.0",

"scripts": {

"ng": "ng",

"start": "ng serve",

"build": "ng build",

"watch": "ng build --watch --configuration development",

"test": "ng test"

},

"private": true,

"dependencies": {

"@angular/animations": "^19.0.0",

"@angular/common": "^19.0.0",

"@angular/compiler": "^19.0.0",

"@angular/core": "^19.0.0",

"@angular/forms": "^19.0.0",

"@angular/platform-browser": "^19.0.0",

"@angular/platform-browser-dynamic": "^19.0.0",

"@angular/router": "^19.0.0",

"rxjs": "~7.8.0",

"tslib": "^2.3.0",

"zone.js": "~0.15.0"

},

"devDependencies": {

"@angular-devkit/build-angular": "^19.0.2",

"@angular/cli": "^19.0.2",

"@angular/compiler-cli": "^19.0.0",

"@types/jasmine": "~5.1.0",

"jasmine-core": "~5.4.0",

"karma": "~6.4.0",

"karma-chrome-launcher": "~3.2.0",

"karma-coverage": "~2.2.0",

"karma-jasmine": "~5.1.0",

"karma-jasmine-html-reporter": "~2.1.0",

"typescript": "~5.6.2"

}

}

Este archivo se modifica cada vez que agregamos nuevos recursos al proyecto, por ejemplo si añadimos 'Angular Material' mediante el comando:

ng add @angular/material

Luego el archivo 'package.json' se ha modificado con la dependencia:

{

"name": "proyecto026",

"version": "0.0.0",

"scripts": {

"ng": "ng",

"start": "ng serve",

"build": "ng build",

"watch": "ng build --watch --configuration development",

"test": "ng test"

},

"private": true,

"dependencies": {

"@angular/animations": "^19.0.0",

"@angular/cdk": "^19.0.1",

"@angular/common": "^19.0.0",

"@angular/compiler": "^19.0.0",

"@angular/core": "^19.0.0",

"@angular/forms": "^19.0.0",

**"@angular/material": "^19.0.1",**

"@angular/platform-browser": "^19.0.0",

"@angular/platform-browser-dynamic": "^19.0.0",

"@angular/router": "^19.0.0",

"rxjs": "~7.8.0",

"tslib": "^2.3.0",

"zone.js": "~0.15.0"

},

"devDependencies": {

"@angular-devkit/build-angular": "^19.0.2",

"@angular/cli": "^19.0.2",

"@angular/compiler-cli": "^19.0.0",

"@types/jasmine": "~5.1.0",

"jasmine-core": "~5.4.0",

"karma": "~6.4.0",

"karma-chrome-launcher": "~3.2.0",

"karma-coverage": "~2.2.0",

"karma-jasmine": "~5.1.0",

"karma-jasmine-html-reporter": "~2.1.0",

"typescript": "~5.6.2"

}

}

Gracias a este archivo que especifica todas las dependencias, luego podemos recrear el código que debe almacenar la carpeta 'node\_modules'.

41 - Vistas diferidas : @defer {} @placeholder{}

Las vistas diferidas es una nueva estructura que se puede emplear en la plantilla HTML y ha sido introducida a partir de Angular 17.

La vistas diferidas permite renderizar un bloque HTML, componente etc. una vez que la página se ha cargado por completo, solo faltando el o los bloques de código diferido.

La carga diferida tiene por objetivo mejorar la experiencia del usuario cuando interactúa con nuestra aplicación.

Sintaxis

La sintaxis más elemental sin parámetros es:

@defer {

aquí disponemos el bloque que se va a renderizar luego que se haya cargado toda la página

} @placeholder {

aquí disponemos el bloque que debe renderizar inmediatamente

}

El bloque @placeholder es opcional pero nos puede ser de mucha utilidad para informar al usuario que se va ha renderizar en dicho lugar, por ejemplo podemos reservar un área de la página donde se mostrará un gráfico estadístico que requiera datos de un servidor web.

Problema

Crear una página con un título, seguidamente muestre los números del 0 al 500000, al final de la página mostrar un pié página con un mensaje.

Como podemos imaginar la generación y renderizado de la página va a ser lenta, debido a su tamaño. Lo más conveniente es que se muestre la página con un mensaje informando al usuario que se está generando la lista de números y tenga un poco de paciencia. Pasemos a crear el proyecto:

ng new proyecto027

En la clase de la componente 'app.component.ts' creamos un arreglo de 500001 elementos con los valores comprendidos entre 0 y 500000:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

numeros = [...Array(500001).keys()];

}

Es importante recordar en JavaScript y por lo tanto aplicable en TypeScript para generar en forma concisa el arreglo:

* Array(500001): Esto crea un nuevo array con una longitud de 500001 elementos. Los elementos en este array aún no tienen valores asignados y serán undefined.
* Array(500001).keys(): La función keys() devuelve un nuevo objeto Array que contiene las claves para cada índice del arreglo. En este caso, las claves serán los números desde 0 hasta 500000.
* [...Array(500001).keys()]: El operador de propagación (...), recordemos que se utiliza para expandir los elementos del objeto Array en un nuevo array. Esto crea un array que contiene los números del 0 al 500000.

En resumen, después de ejecutar este código, la variable numeros contendrá un array con los números del 0 al 500000.

Ahora veamos como creamos el bloque que generará la lista de números en forma diferible, pasamos a codificar en el archivo app.component.html:

<h1>Listado de numeros del 0 al 500000</h1>

@defer {

<p>

@for(num of numeros;track num) {

{{num}} -

}

</p>

}

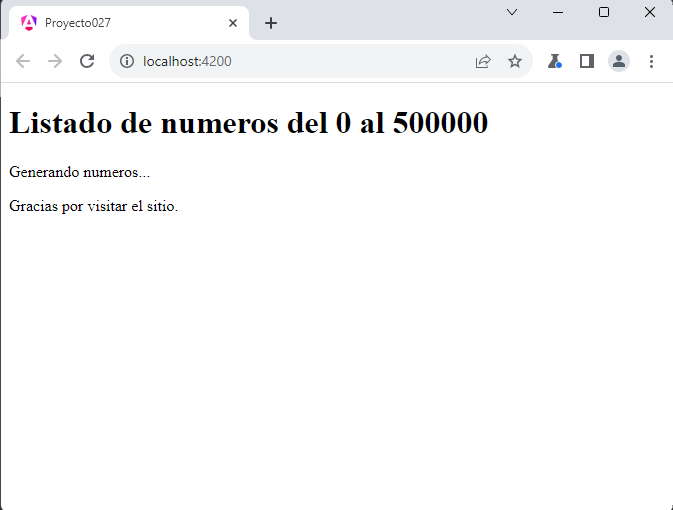
@placeholder {

<p>Generando numeros...</p>

}

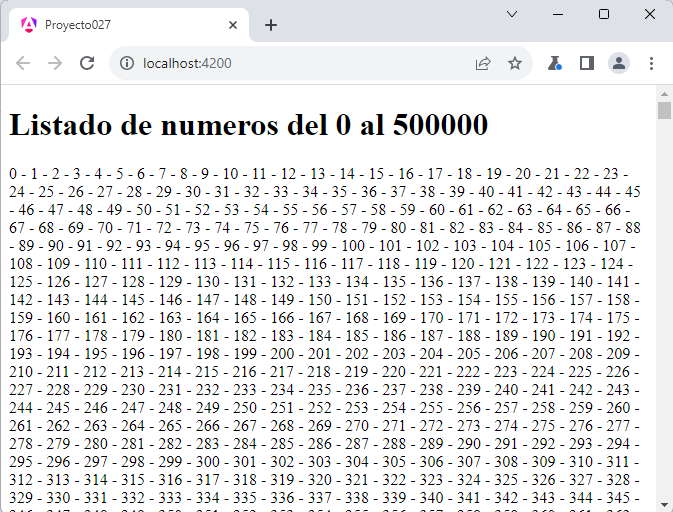
<p>Gracias por visitar el sitio.</p>

Podemos ejecutar la aplicación y ver que inmediatamente se carga la página casi por completo, pero queda pendiente el bloque @defer:



Mientras no se ha resuelto el contenido del bloque @defer se muestra el contenido del bloque @placeholder.

Luego de algunos segundos, dependiendo de la velocidad de la computadora se remplaza el contenido del @placeholder por el contenido del bloque @defer, en nuestro ejemplo mostrando la lista de números en pantalla :



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto027/).

Sin vista diferida podemos comprobar que la página tarda varios segundos antes que se muestre en el navegador, lo cual genera una experiencia de usuario muy pobre. Puede modificar y probar el código sin el bloque de vista diferida:

<h1>Listado de numeros del 0 al 500000</h1>

<p>

@for(num of numeros;track num) {

{{num}} -

}

</p>

<p>Generando numeros...</p>

<p>Gracias por visitar el sitio.</p>

Seguramente sin vista diferida deberíamos utilizar otro algoritmo para que muestre parte de la página hasta que se muestre la lista de números.

Si utilizamos una versión de Angular anterior a la 17, un algoritmo posible es el siguiente:

import { Component, OnInit } from '@angular/core';

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

standalone: true,

imports: [CommonModule, RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

numeros = [...Array(500001).keys()];

lista = 'generando la lista...';

ngOnInit() {

setTimeout(() => {

this.lista = this.numeros.join('-');

}, 0);

}

}

OnInit es una interfaz que proporciona el método ngOnInit, que se ejecuta cuando la componente se inicializa.

El método ngOnInit() se llama después de que la componente ha sido inicializada. En este caso, se utiliza para actualizar la propiedad lista después de un breve retraso, simbolizado por setTimeout. Este enfoque se utiliza para mostrar el mensaje 'generando la lista...' antes de que la lista completa de números se procese y actualice en la vista.

Y su plantilla queda con el siguiente código:

<h1>Listado de numeros del 0 al 500000</h1>

{{lista}}

<p>Gracias por visitar el sitio.</p>

La vista diferida nos permite implementar un código más legible y veremos más adelante que tiene otras ventajas.

42 - Vistas diferidas : @defer ([opciones de activación]) {}

El bloque diferible podemos indicar bajo que circunstancias debe cargarse:

on idle: La opción de activación por defecto. Por lo mismo es que no se la agrega:

@defer (on idle){

<p>

@for(num of numeros;track num) {

{{num}} -

}

</p>

}

En este caso lo más común es no disponer dicha activación:

@defer {

<p>

@for(num of numeros;track num) {

{{num}} -

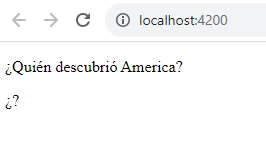
}

</p>

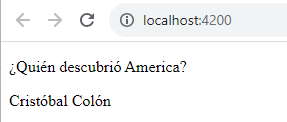
}

* on timer: se activaría después de una duración específica. La duración es obligatoria y se puede especificar en 'ms' o 's':
* <p>¿Quién descubrió America?</p>
* @defer (on timer(5s)) {
* <p>Cristóbal Colón</p>
* } @placeholder {
* <p>¿?</p>
* }

Primero se muestra el contenido del bloque @placeholder y luego de 5 segundos es remplazado por el contenido del bloque @defer.

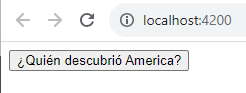


Y luego de 5 segundos:

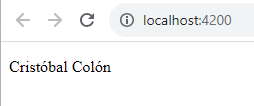


* on interaction: Se activará el bloque diferido cuando el usuario interactúe con el elemento especificado en el @placeholder a través de los eventos click o keydown.
* @defer (on interaction) {
* <p>Cristóbal Colón</p>
* } @placeholder {
* <button>¿Quién descubrió America?</button>
* }

Primero se muestra el botón:



Y al ser presionado el botón aparece el párrafo remplazando el contenido del @placeholder:



* on hover: se activa la carga diferida cuando el mouse se coloca sobre el área de activación. Los eventos utilizados para esto son mouseenter y focusin.
* @defer (on hover) {
* <p>Cristóbal Colón</p>
* } @placeholder {
* <p>¿Quién descubrió America?</p>
* }

Si desplazamos la flecha del mouse sobre el párrafo '¿Quién descubrió America?', el mismo será actualizado por el párrafo 'Cristóbal Colón'.

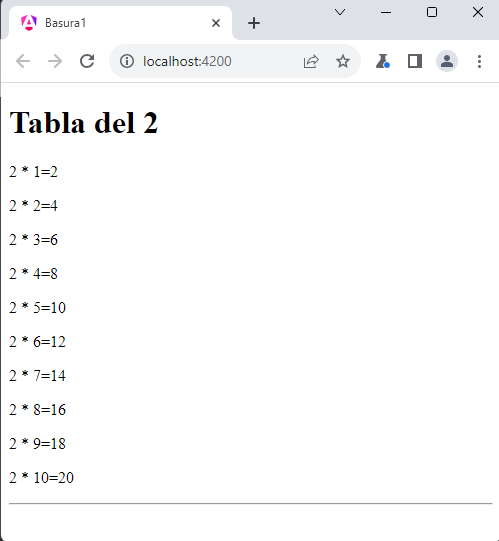
* on viewport: se activa el bloque diferido cuando el contenido se encuentra en la ventana visible del navegador. Es muy útil si tenemos páginas que requieren un scroll casi infinito. No tenemos que renderizar si no avanzamos con el scroll de la página. El siguiente ejemplo que vamos a realizar utilizará esta opción de activación de la vista diferida.

Problema

Crear una componente llamada 'tablamultiplicacion' que reciba en una propiedad el número del cual queremos que muestre la tabla de multiplicación y se encargue de mostrarla por pantalla.

<app-tablamultiplicacion [tabla]="2" />

El resultado de visual de esta componente debe generar la siguiente pantalla:



Por otro lado la componente principal de la aplicación generar las tablas de multiplicar del 2 al 10000. Mediante vistas diferidas evitar que se generen todas las tablas en forma inmediata y haga que la experiencia de usuario sea aceptable.

Creemos el proyecto:

ng new proyecto028

Primero creamos la componente:

ng generate component tablamultiplicacion

Codificamos la clase 'tablamultiplicacion.component.ts':

import { Component, Input } from '@angular/core';

@Component({

selector: 'app-tablamultiplicacion',

imports: [],

templateUrl: './tablamultiplicacion.component.html',

styleUrl: './tablamultiplicacion.component.css'

})

export class TablamultiplicacionComponent {

@Input() tabla:number=0;

}

Por otro lado la vista 'tablamultiplicacion.component.html':

<h1>Tabla del {{tabla}}</h1>

@for(valor of [1,2,3,4,5,6,7,8,9,10];track $index) {

<p>

{{tabla}} \* {{valor}}={{tabla\*valor}}

</p>

}

<hr>

Pasemos ahora a modificar la componente creada por defecto 'app.component.ts':

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

import { RouterOutlet } from '@angular/router';

import { TablamultiplicacionComponent } from './tablamultiplicacion/tablamultiplicacion.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, TablamultiplicacionComponent],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

numeros = [...Array(9999).keys()].map(num => num + 2);

}

Creamos un arreglo llamado numeros donde guardamos los valores del 2 al 10000:

Por último la plantillo donde hacemos uso de las vistas diferidas 'app.component.html':

@for(valor of numeros;track valor) {

@defer (on viewport) {

<app-tablamultiplicacion [tabla]="valor" />

}

@placeholder {

<p>Esperando la tabla del {{valor}}</p>

}

}

<router-outlet />

Dentro del ciclo @for creamos las 9999 vistas diferidas y su activación depende que se esté viendo en la pantalla del navegador, como podemos imaginar la mayoría de las tablas están fuera del viewport del navegador (según el tamaño de la pantalla pueden estar visibles 2 o tres tablas por pantalla)

La experiencia de ejecución es aceptable gracias a que utilizamos la activación por viewport.

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto028/).

Probemos de eliminar el 'on viewport' y veremos que el navegador se toma su tiempo en renderizar las 9999 tablas de multiplicar:

@for(valor of numeros;track valor) {

@defer {

<app-tablamultiplicacion [tabla]="valor" />

}

@placeholder {

<p>Esperando la tabla del {{valor}}</p>

}

}

43 - Vistas diferidas : Lazy Loading (carga perezosa)

Las vistas diferidas nos permite en algunas situaciones el concepto de Lazy Loading al permitir organizar una aplicación Angular que no cargue en forma completa toda la aplicación web en una única llamada al servidor, sino que retrase la carga hasta el momento de su utilización.

El proceso de particionar la aplicación en distintos archivos JavaScript lo hace automáticamente Angular cuando construimos la aplicación.

Veamos un problema y verifiquemos en el navegador que cuando disponemos una o más componentes en un bloque @defer, el mismo se almacena en un archivo JavaScript separado y es peticionado al servidor en el momento que sea necesario.

Problema

Crear una componente que llamada 'graficotarta' que muestre un gráfico de tarta genérico. Luego en la componente principal de nuestra aplicación proceder a mostrar el gráfico cuando hagamos click sobre un botón.

Creemos el proyecto:

ng new proyecto029

Primero creamos la componente:

ng generate component graficotarta

No hacemos ningún cambio al archivo 'graficotarta.component.ts':

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

@Component({

selector: 'app-graficotarta',

imports: [],

templateUrl: './graficotarta.component.html',

styleUrl: './graficotarta.component.css'

})

export class GraficotartaComponent {

}

En la plantilla 'graficotarta.component.html' disponemos un div:

<div class="tarta"></div>

En la hoja de estilo 'graficotarta.component.css' disponemos:

.tarta {

width: 200px;

height: 200px;

background: conic-gradient(

#ff5733 0% 33%, /\* Color para la primera porción \*/

#33ff57 33% 66%, /\* Color para la segunda porción \*/

#5733ff 66% 100% /\* Color para la tercera porción \*/

);

border-radius: 50%;

}

Pasemos ahora a modificar la componente creada por defecto 'app.component.ts' importando la otra componente:

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

import { RouterOutlet } from '@angular/router';

import { GraficotartaComponent } from './graficotarta/graficotarta.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, GraficotartaComponent],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'proyecto029';

}

y la plantilla 'app.component.html' donde cargaremos mediante una vista diferida el gráfico de tarta:

@defer (on interaction) {

<app-graficotarta></app-graficotarta>

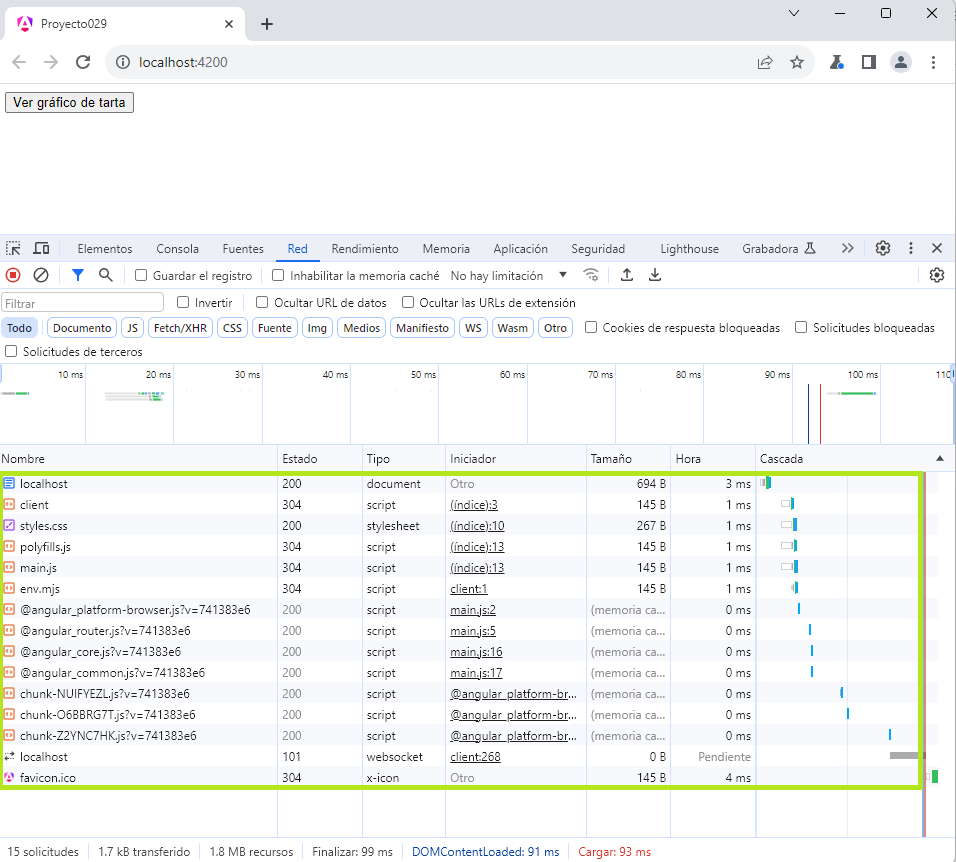
} @placeholder {

<button>Ver gráfico de tarta</button>

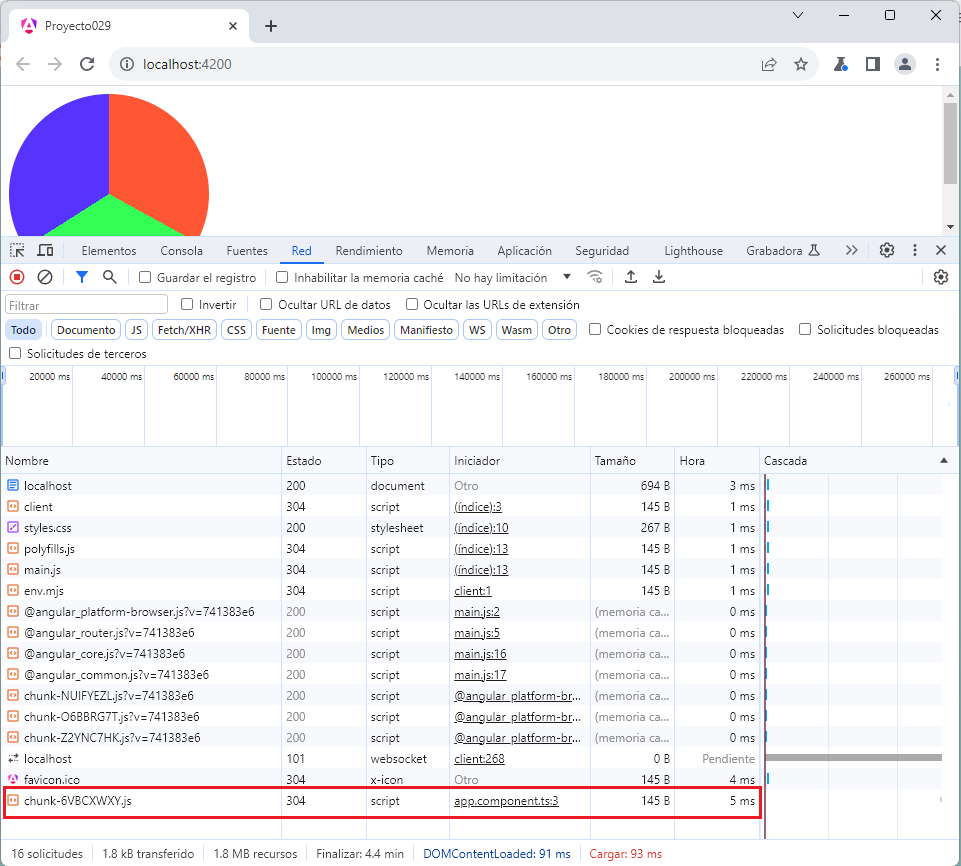
}

<router-outlet />

Lo más interesante es ver que sucede en las peticiones de archivos desde el navegador. Debemos abrir la ventana de herramientas del navegador y en la pestaña 'Red' podemos observar los archivos solicitados al servidor hasta ese momento:



Luego presionamos el botón y podemos observar que se solicita otro archivo al servidor con el código que genera el gráfico de tarta:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto029/).

En una aplicación grande podemos dividir distintas funcionalidades y realizar su carga con lazy loading.

comando prefetch (captación previa)

Vimos en el ejemplo anterior que con la carga perezosa podemos reducir la sobrecarga del servidor, por ejemplo si el usuario no presiona el botón no será necesario solicitar al servidor la componente.

Hay situaciones donde tal vez lo más importante es la velocidad de ejecución de nuestra aplicación y no queremos que el usuario cuando presione el botón tenga que esperar la recuperación de datos del servidor, para esto disponemos un comando llamado 'prefetch':

@defer (on interaction; prefetch on idle) {

<app-graficotarta></app-graficotarta>

} @placeholder {

<button>Ver gráfico de tarta</button>

}

Agregando 'prefetch on idle', con esto indicamos que el bloque se recupere del servidor sin la necesidad que se dispare el 'on interaction'.

44 - Vistas diferidas : @defer (when [expresión lógica]) {}

Otra variante que podemos implementar con las vistas diferidas es utilizando la palabra clave when con una expresión lógica que retorne true o false. La vista se materializará cuando la expresión lógica se verifique verdadera.

Creemos un proyecto y veamos el funcionamiento de esta variante de vista diferida:

ng new proyecto030

En el archivo 'app.component.ts' definimos un atributo que almacene el valor false:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

visible = false;

}

La la plantilla 'app.component.html' queda con el siguiente código:

<p>¿Quién descubrió America?</p>

<input type="button" (click)="visible=true" value="mostrar respuesta">

@defer (when visible) {

<p>Respuesta:Cristóbal Colón</p>

} @placeholder {

<p>¿?</p>

}

<router-outlet />

Como el atributo visible almacena el valor false, el bloque de código no se renderiza. Luego cuando presionamos el botón cambiamos el valor del atributo visible por el valor false, en dicho momento se hace visible el bloque de código definido en el @defer.

Tener en cuenta que en una aplicación más realista en el bloque @defer tendremos una componente, que se cargará con lazy loading una vez que el when se verifique con el valor true.

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto030/).

Podemos utilizar también el prefetch, para el caso en que sea una componente su contenido se recupere del servidor inmediatamente y cuando el when se verifique verdadero, la componente se renderize inmediatamente:

@defer (when visible;prefetch on idle) {

<p>Respuesta:Cristóbal Colón</p>

} @placeholder {

<p>¿?</p>

}

45 - Vistas diferidas : @defer {} @placeholder{} @loading{} @error{}

Hemos dicho que cuando definimos una vista diferida, el único bloque obligatorio es el @defer {}.

A parte del **@defer** podemos agregar:

* **@placeholder** : Ya dijimos y hemos trabajado con este bloque que aparece mientras la vista no ha sido activada. Este contenido se reemplaza con el contenido principal una vez que se completa la carga.
* **@loading** : El bloque @loadinge es un bloque opcional que le permite declarar el contenido que se mostrará durante la carga de del bloque @defer. Por ejemplo, podría mostrar una rueda giratoria de carga mientras el contenido se recupera de un servidor.
* @defer {
* <componente></componente>
* } @loading (after 100ms; minimum 1s) {
* <img alt="cargando..." src="cargando.gif" />
* }

El bloque @loading acepta dos parámetros opcionales para especificar la mínima cantidad de tiempo que se debe mostrar el bloque y la cantidad de tiempo que se debe esperar para que comience la carga antes de mostrar la plantilla de carga. minimum y after los parámetros se especifican en incrementos de tiempo de milisegundos (ms) o segundos (s).

Al igual que @placeholder, estos parámetros existen para evitar el parpadeo rápido del contenido en el caso de que las dependencias diferidas se recuperen rápidamente. Tanto los temporizadores minimum como after para el bloque @loading comienzan inmediatamente después de que se haya activado la carga del @defer.

* **@error** : El bloque @error permite declarar contenido que se mostrará si falla la carga diferida.
* @defer {
* <componente></componente>
* } @loading (after 100ms; minimum 1s) {
* <img alt="cargando..." src="cargando.gif" />
* } @error {
* <p>Problemas con el servidor</p>
* }

Imaginemos que se corta la conexión con el servidor, cuando se trate de recuperar el archivo que almacena <componente></componente>, se dispara el bloque @error.

46 - Directiva de atributo [ngStyle] y directivas para estilos individuales - Sintaxis completa

La directiva de atributo ngStyle actualiza los estilos para un elemento HTML determinado.

Establece una o más propiedades de estilo, especificadas como pares clave-valor separados por dos puntos. La clave es un nombre de estilo y el valor es una expresión a evaluar o valor a asignar.

Podemos crear un proyecto para ir probando estas funcionalidades:

ng new proyecto031

Si indicamos directamente el valor a asignar tenemos la siguiente sintaxis:

<h1 [ngStyle]="{'color':'red','background-color':'#ff0','text-align':'center'}">Sitio fuera de servicio</h1>

En el ejemplo vemos que si asignamos valores no presenta ninguna ventaja que definir la propiedad 'style' de HTML:

<h1 style="color:red;background-color:#ff0;text-align:center">Sitio fuera de servicio</h1>

Pero cuando utilizamos la directiva de atributo ngStyle podemos indicar una expresión en la zona del valor, por ejemplo si tenemos definido el atributo en la clase:

colorEstado='#f00';

Luego en la vista accedemos a dicho atributo:

<h1 [ngStyle]="{'color':colorEstado}">Sitio fuera de servicio</h1>

Estamos indicando que para el atributo 'color' debe tomar el valor almacenado en la variable colorEstado, es importante notar que no debe ir entre comillas la variable.

Podemos llamar a un método en el lugar de la expresión, si tenemos definido en el modelo el método:

retornarColor() {

return '#00f';

}

Luego cuando definimos la directiva tenemos la sintaxis:

<h1 [ngStyle]="{'color':retornarColor()}">Sitio fuera de servicio</h1>

Disponer una condición en la expresión

Por ejemplo si necesitamos que una tabla que muestra los números del 1 al 5, tengan las celdas de colores alternos, lo podemos resolver con la sintaxis:

<table>

@for(elemento of [1,2,3,4,5];track $index) {

<tr>

<td [ngStyle]="{'background-color':elemento%2==0?'red':'yellow'}">{{elemento}}</td>

</tr>

}

</table>

Cada vuelta del for se verifica si el resto de dividir el elemento del array es cero procede a asignar el valor 'red' a la propiedad 'background-color' en caso contrario asigna 'yellow'.

Trabajar con unidades de medida.

La sintaxis a emplear con unidades de medida se deben indicar como sufijo cuando la definimos:

@for(elemento of [10,12,14,20,30];track $index) {

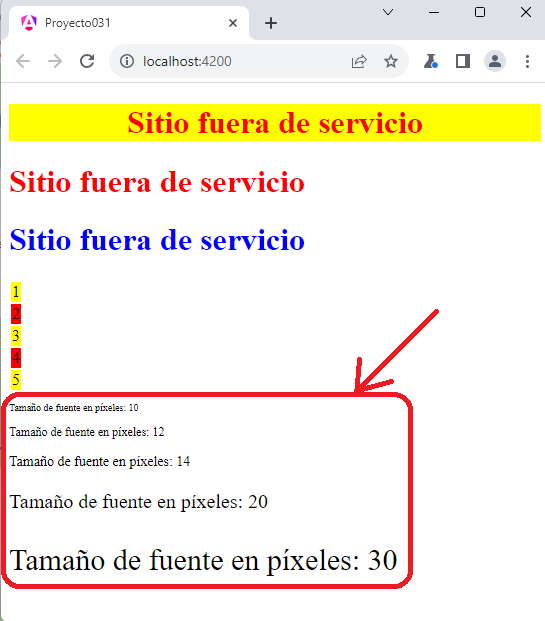
<p [ngStyle]="{'font-size.px':elemento}">

Tamaño de fuente en píxeles: {{elemento}}

</p>

}

Luego tenemos como resultado:



Es decir utilizamos la sintaxis: 'font-size.px'

Otros ejemplos de unidades de medida asignada podrían ser:

* 'margin-top.px':'10'
* 'padding.em':'1'

Asignar a la directiva un objeto con un conjunto de estilos

Podemos crear en el modelo un objeto literal con distintos estilos, que luego podemos modificar cuando se producen eventos. Para probar esto crearemos un div con una etiqueta y dos botones que modifican el tamaño de la fuente.

En el modelo de la componente definimos 'app.component.ts':

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

import { RouterOutlet } from '@angular/router';

import { NgStyle } from '@angular/common';

@Component({

selector: 'app-root',

imports: [RouterOutlet, NgStyle],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

colorEstado = '#f00';

retornarColor() {

return '#00f';

}

tamano = 30;

presentacion = {

"background-color": "black",

"color": "white",

"width.px": "1000",

"height.px": "200",

"font-size.px": this.tamano,

"display": "flex",

"justify-content": "center",

"align-items": "center"

}

agrandar() {

this.tamano++;

this.presentacion["font-size.px"] = this.tamano;

}

reducir() {

this.tamano--;

this.presentacion["font-size.px"] = this.tamano;

}

}

En la vista tenemos 'app.component.html':

<h1 [ngStyle]="{'color':'red','background-color':'#ff0','text-align':'center'}">Sitio fuera de servicio</h1>

<h1 [ngStyle]="{'color':colorEstado}">Sitio fuera de servicio</h1>

<h1 [ngStyle]="{'color':retornarColor()}">Sitio fuera de servicio</h1>

<table>

@for(elemento of [1,2,3,4,5];track $index) {

<tr>

<td [ngStyle]="{'background-color':elemento%2==0?'red':'yellow'}">{{elemento}}</td>

</tr>

}

</table>

@for(elemento of [10,12,14,20,30];track $index) {

<p [ngStyle]="{'font-size.px':elemento}">

Tamaño de fuente en píxeles: {{elemento}}

</p>

}

<div [ngStyle]="presentacion">Texto centrado en forma horizontal y vertical.</div>

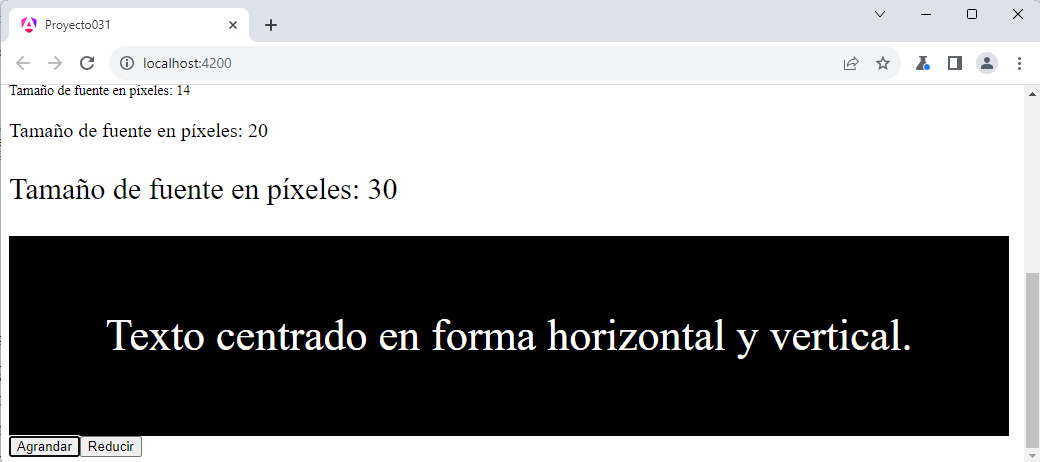
<button (click)="agrandar()">Agrandar</button>

<button (click)="reducir()">Reducir</button>

<router-outlet />

Hemos asignado a la directiva ngStyle la variable 'presentación' que se define en el modelo.

Cuando se presiona alguno de los botones procedemos a actualizar la variable 'presentacion' y podemos ver que el texto se actualiza.



Sintaxis para directivas individuales

Podemos hacer referencia a una propiedad individual con la sintaxis:

<p [style.color]="'#f00'">Color rojo</p>

Disponemos 'style' y seguidamente el nombre de la propiedad, en este caso 'color'. Son importante las comillas simples dentro de las comillas dobles, estamos indicando que tomo el string '#f00'.

Es más común que enlacemos la directiva con una variable definida en el modelo, por ejemplo si tenemos en el modelo:

colorFondo='#ff0';

Luego en la vista enlazamos esta variable con la sintaxis:

<p [style.background-color]="colorFondo">Color</p>

Agregar a la aplicación anterior la posibilidad de modificar el tamaño de la fuente de un texto mediante dos botones.

En el archivo 'app.component.ts' tenemos:

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

import { RouterOutlet } from '@angular/router';

import { NgStyle } from '@angular/common';

@Component({

selector: 'app-root',

imports: [RouterOutlet, NgStyle],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

colorEstado = '#f00';

retornarColor() {

return '#00f';

}

tamano = 30;

presentacion = {

"background-color": "black",

"color": "white",

"width.px": "1000",

"height.px": "200",

"font-size.px": this.tamano,

"display": "flex",

"justify-content": "center",

"align-items": "center"

}

agrandar() {

this.tamano++;

this.presentacion["font-size.px"] = this.tamano;

}

reducir() {

this.tamano--;

this.presentacion["font-size.px"] = this.tamano;

}

tamanoFuente = 50;

agrandarFuente() {

this.tamanoFuente++;

}

reducirFuente() {

this.tamanoFuente--;

}

}

En el archivo 'app.component.html' queda:

<h1 [ngStyle]="{'color':'red','background-color':'#ff0','text-align':'center'}">Sitio fuera de servicio</h1>

<h1 [ngStyle]="{'color':colorEstado}">Sitio fuera de servicio</h1>

<h1 [ngStyle]="{'color':retornarColor()}">Sitio fuera de servicio</h1>

<table>

@for(elemento of [1,2,3,4,5];track $index) {

<tr>

<td [ngStyle]="{'background-color':elemento%2==0?'red':'yellow'}">{{elemento}}</td>

</tr>

}

</table>

@for(elemento of [10,12,14,20,30];track $index) {

<p [ngStyle]="{'font-size.px':elemento}">

Tamaño de fuente en píxeles: {{elemento}}

</p>

}

<div [ngStyle]="presentacion">Texto centrado en forma horizontal y vertical.</div>

<button (click)="agrandar()">Agrandar</button>

<button (click)="reducir()">Reducir</button>

<div [style.font-size.px]="tamanoFuente">Texto</div>

<button (click)="agrandarFuente()">Agrandar</button>

<button (click)="reducirFuente()">Reducir</button>

<router-outlet />

Como vemos si tenemos que trabajar con unidades de medida debemos indicarla en la misma directiva:

<div [style.font-size.px]="tamano">Texto</div>

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto031/).

Acotaciones

Debemos usar NgStyle cuando debemos establecer muchos estilos en línea de forma simultánea y dinámica, según el estado del componente, en el caso que solo debemos establecer un único estilo conviene utilizar la sintaxis que acabamos de ver.

47 - Directiva de atributo [ngClass] y directivas para clases individuales - Sintaxis completa

En el concepto anterior vimos las diferentes herramientas que nos provee Angular para manipular los estilos en línea. Ahora veremos que Angular nos permite manipular las clases que se asocian a un elemento HTML.

Podemos crear un proyecto para ir probando estas funcionalidades:

ng new proyecto032

La directiva de atributo [ngClass] permite establecer una o más clases a un elemento HTML.

Veamos distintas formas de utilizar la directiva [ngClass]

Si tenemos definidas las siguientes clases en el archivo 'app.component.css':

.clase1 {

color:red;

background-color:blue;

}

.clase2 {

font-size:20px;

font-family: 'Courier New', Courier, monospace;

}

Luego en la vista 'app.component.html':

<p [ngClass]="'clase1 clase2'">Prueba de directiva ngClass</p>

Se le asignan las clases 'clase1' y 'clase2' al párrafo. Son importante las comillas simples para que funcione. Normalmente no haríamos esto ya que es más sencillo fijar el atributo 'class' de HTML:

<p class="clase1 clase2">Prueba de class</p>

Una alternativa más útil es definir un objeto literal cuyas claves son nombres de clases y sus valores un valor boolean que indica si se debe activar o no la clase al elemento HTML.

Si tenemos los atributos:

estado1=true;

estado2=false;

Luego en la vista definimos la directiva de atributo [ngClass]:

<p [ngClass]="{'clase1':estado1,'clase2':estado2}">Prueba de directiva ngClass</p>

Luego solo se activa la 'clase1' ya que 'estado1' almacena true, debido a que 'estado2' almacena false significa que la 'clase2' no se aplica a la etiqueta.

Como vemos podemos en forma dinámica cambiar variables definidas en el modelo y se aplican los cambios gracias a la directiva [ngClass]

Disponer una condición en la expresión

Por ejemplo si necesitamos que una tabla que muestra los números del 1 al 5, tengan las celdas de colores alternos, lo podemos resolver con la sintaxis:

<table>

@for(elemento of [1,2,3,4,5];track $index) {

<tr>

<td [ngClass]="{'color1':elemento%2==0,'color2':elemento%2==1}">{{elemento}}</td>

</tr>

}

</table>

Suponiendo que las dos clases están definidas en el archivo 'app.component.css':

.color1 {

color: white;

background-color: blue;

}

.color2 {

color: white;

background-color: green;

}

Cada vuelta del for solo se aplica una de las dos clases.

Vamos a agregar a la aplicación otra funcionalidad que permita mediante dos botones fijar o eliminar dos estilos que se aplican a un elemento HTML.

app.component.css

.clase1 {

color:red;

background-color:blue;

}

.clase2 {

font-size:20px;

font-family: 'Courier New', Courier, monospace;

}

.color1 {

color: white;

background-color: blue;

}

.color2 {

color: white;

background-color: green;

}

.clase3 {

color:red;

background-color:blue;

}

.clase4 {

font-size:20px;

font-family: 'Courier New', Courier, monospace;

}

app.component.html

<p [ngClass]="'clase1 clase2'">Prueba de directiva ngClass</p>

<p [ngClass]="{'clase1':estado1,'clase2':estado2}">Prueba de directiva ngClass</p>

<table>

@for(elemento of [1,2,3,4,5];track $index) {

<tr>

<td [ngClass]="{'color1':elemento%2==0,'color2':elemento%2==1}">{{elemento}}</td>

</tr>

}

</table>

<p [ngClass]="forma">Prueba de directiva</p>

<button (click)="fijar()">Fijar clases</button>

<button (click)="eliminar()">Eliminar clases</button>

app.component.ts

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

import { RouterOutlet } from '@angular/router';

import { NgClass } from '@angular/common';

@Component({

selector: 'app-root',

imports: [RouterOutlet, NgClass],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

estado1 = true;

estado2 = false;

forma = {

'clase3 clase4': true

};

fijar() {

this.forma['clase3 clase4'] = true;

}

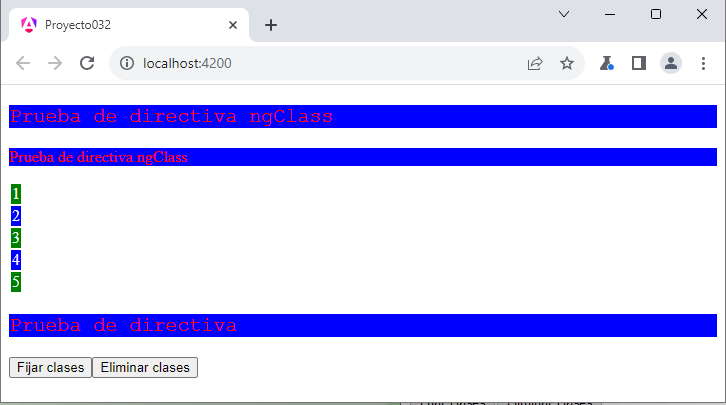
eliminar() {

this.forma['clase3 clase4'] = false;

}

}

Se asocia la variable 'forma' a la directiva 'ngClass', luego significa que cuando comienza la ejecución de la aplicación las dos clases se asocian a dicha etiqueta:



Luego si se presiona el botón de 'eliminar', modificamos el objeto literal asignando el valor false, esto hace que las dos clases se eliminen de la etiqueta.

Enlace a clases individuales.

En lugar de utilizar la directiva 'ngClass' podemos utilizar una sintaxis alternativa para agregar o eliminar clases en particular para una etiqueta HTML:

<p [class.clase1]="true">Prueba de class</p>

Se añade las 'clase1' al parrafo. En lugar de disponer un true podemos definir una condición o llamar a una función que retorne true o false:

<table>

@for(elemento of [1,2,3,4,5];track $index) {

<tr>

<td [class.clase1]="elemento%2==0">{{elemento}}</td>

</tr>

}

</table>

Con el código anterior solo se incorpora la 'clase1' a las filas pares de la tabla. Si la etiqueta ya dispone de otras clases no se borran las mismas

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto032/).

48 - Directivas de atributo - creación de directivas personalizadas

Hemos dicho que una directiva de atributo cambia la apariencia o el comportamiento de un elemento HTML. Hay varias directivas que trae Angular por defecto. Ahora veremos que podemos crear nuestras propias directivas de atributo personalizadas y definir el algoritmo que se debe aplicar al elemento HTML que se le asigna.

La herramienta de Angular CLI provee lo necesario para crear un esqueleto básico de una directiva de atributo. Debemos utilizar la siguiente sintaxis:

ng generate directive [nombre de la directiva]

Confeccionaremos un problema elemental para conocer los pasos en la creación, codificación y uso de una directiva de atributo personalizada.

Problema

Crear una directiva personalizada que se pueda asociar a cualquier elemento HTML y cuyo objetivo sea resaltar el texto que muestra, cambiando el color de fondo por amarillo.

* Crearemos primero el proyecto
* ng new proyecto033
* Procedemos a crear la directiva de atributo personalizada llamando a la misma 'resaltado':
* ng generate directive resaltado

Se crean dos archivos:

creación de directiva Angular

Se crea propiamente el archivo que contendrá la lógica de la directiva y tiene como nombre 'resaltado.directive.ts':

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

@Directive({

selector: '[appResaltado]'

})

export class ResaltadoDirective {

constructor() { }

}

También se crea el archivo 'resaltado.directive.spec.ts' para especificar pruebas unitarias (por el momento no hemos trabajado con este tipo de archivos, no lo modificaremos ni analizaremos):

import { ResaltadoDirective } from './resaltado.directive';

describe('ResaltadoDirective', () => {

it('should create an instance', () => {

const directive = new ResaltadoDirective();

expect(directive).toBeTruthy();

});

});

* Procedemos a modificar el archivo 'resaltado.directive.ts' implementando la lógica de nuestra directiva:
* import { Directive, ElementRef } from '@angular/core';
* @Directive({
* selector: '[appResaltado]'
* })
* export class ResaltadoDirective {
* constructor(private elemento: ElementRef) {
* elemento.nativeElement.style.backgroundColor = 'yellow';
* }
* }

Se inyecta al constructor un objeto de la clase 'ElementRef':

constructor(private elemento: ElementRef) {

Previamente debemos importar la clase 'ElementRef', que se almacena en '@angular/core':

import { Directive, ElementRef } from '@angular/core';

La lógica de nuestra directiva es muy simple, solo debe manipular el color de fondo del elemento HTML al que se le aplicó.

El parámetro 'elemento' tiene la referencia al elemento HTML que se le aplicó la directiva, mediante el método 'nativeElement' podemos acceder a los atributos y métodos del DOM.

* Solo nos falta importar la directiva en nuestra componente para luego poder consumir la directiva de atributo en la interfaz visual de una componente. Pasemos a modificar el archivo 'app.component.ts' por:
* import { Component } from '@angular/core';
* import { RouterOutlet } from '@angular/router';
* import { ResaltadoDirective } from './resaltado.directive';
* @Component({
* selector: 'app-root',
* imports: [RouterOutlet, ResaltadoDirective],
* templateUrl: './app.component.html',
* styleUrls: ['./app.component.css']
* })
* export class AppComponent {
* title = 'proyecto033';
* }

Luego el archivo 'app.component.html' consumimos la directiva de atributo:

<h3>Datos generales.</h3>

<p>Chile es un país de América ubicado en el extremo sudoeste de América del Sur.

Su nombre oficial es <span appResaltado>República de Chile</span>

y su <span appResaltado>capital es la ciudad de Santiago</span>.

Primer país sudamericano en ingresar a la Organización para la Cooperación y el

Desarrollo Económicos, Chile es una de las economías de América Latina que más ha

crecido desde mediados de la década de 1980.</p>

<p>Antes del descubrimiento de América, las tierras situadas al sur del desierto de Atacama ya

se llamaban <span appResaltado>Chili</span> en la tradición indígena.

Una vez instalados en Nueva Castilla y

Nueva Toledo, los conquistadores españoles siguieron llamando de esa forma a la

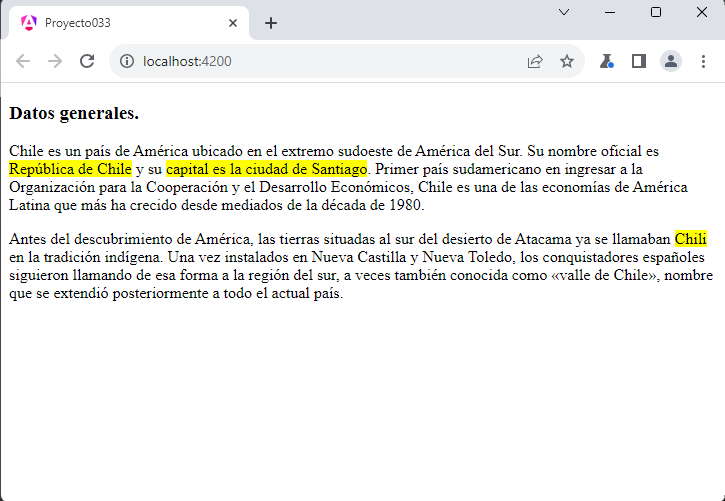
región del sur, a veces también conocida como «valle de Chile», nombre que se extendió

posteriormente a todo el actual país.</p>

<router-outlet />

Cuando queremos resaltar un texto, podemos utilizar el elemento 'span' y asignar la directiva 'appResaltado'.

Tenemos como resultado en el navegador:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto033/).

Acotaciones

* Podemos aplicar la directiva creada a cualquier otro elemento HTML:
* <h1 appResaltado>Estado de resultados</h1>
* <p appResaltado>Esto es una prueba.</p>
* Si queremos que la directiva tenga un prefijo distinto a 'app' cuando se crea, debemos utilizar la sintaxis:
* ng generate directive resaltado --prefix lib
* Si queremos que la directiva no tenga prefijo cuando se crea, debemos utilizar la sintaxis:

ng generate directive resaltado --prefix

49 - Directivas de atributo - creación y definición de propiedades

Vimos en el concepto anterior los pasos que debemos dar para crear una simple directiva. Ahora veremos que podemos pasar uno o más datos para personalizar la directiva con datos a enviarle.

Problema

Crear una directiva personalizada que se pueda asociar a cualquier elemento HTML y cuyo objetivo sea resaltar el texto que muestra, cambiando el color de fondo por amarillo por defecto o un color que le pasemos a dicha directiva. Agregar un segundo dato a enviar a la directiva que sea el tamaño de la fuente.

* Crearemos primero el proyecto
* ng new proyecto034
* Procedemos a crear la directiva de atributo personalizada llamando a la misma 'resaltado':
* ng generate directive resaltado

Se crean dos archivos.

Se crea propiamente el archivo que contendrá la lógica de la directiva y tiene como nombre 'resaltado.directive.ts':

También se crea el archivo 'resaltado.directive.spec.ts' para especificar pruebas unitarias (por el momento no hemos trabajado con este tipo de archivos, no lo modificaremos ni analizaremos)

* Procedemos a modificar el archivo 'resaltado.directive.ts' implementando la lógica de nuestra directiva:
* import { Directive, ElementRef, Input, OnInit, SimpleChanges, OnChanges } from '@angular/core';
* @Directive({
* selector: '[appResaltado]',
* })
* export class ResaltadoDirective implements OnInit, OnChanges {
* @Input('appResaltado') colorResaltado!: string;
* @Input('tamano') tam: number = 0;
* constructor(private elemento: ElementRef) {
* }
* ngOnInit(): void {
* this.actualizar();
* }
* ngOnChanges(changes: SimpleChanges) {
* this.actualizar();
* }
* actualizar() {
* if (this.colorResaltado != null)
* this.elemento.nativeElement.style.backgroundColor = this.colorResaltado;
* else
* this.elemento.nativeElement.style.backgroundColor = 'yellow';
* if (this.tam > 0)
* {
* console.log(this.elemento.nativeElement);
* this.elemento.nativeElement.style.fontSize = this.tam + 'px';
* }
* }
* }

Se inyecta al constructor un objeto de la clase 'ElementRef':

constructor(private elemento: ElementRef) {

Debemos importar 6 clases, que se almacenan en '@angular/core':

import { Directive, ElementRef, Input, OnInit, SimpleChanges, OnChanges } from '@angular/core';

El método onInit que se ejecuta una vez que la directiva ha sido creada y almacenado los datos en las propiedades appResaltado y tamano. Desde el método onInit llamamos al método 'actualizar'.

El método actualizar modifica el color de fondo del elemento HTML según el dato recibido en la propiedad 'appResaltado' o en el caso que no lo recibe pinta el elemento de amarillo:

actualizar() {

if (this.colorResaltado != null)

this.elemento.nativeElement.style.backgroundColor = this.colorResaltado;

else

this.elemento.nativeElement.style.backgroundColor = 'yellow';

También en el método actualizar si hemos inicializado la propiedad 'tamano' procedemos a modificar el tamaño de la fuente:

if (this.tam > 0)

this.elemento.nativeElement.style.fontSize = this.tam + 'px';

El método ngOnChanges se dispara si cambiamos el valor de una propiedad de la directiva, en cuyo caso procedemos a actualizar el color de fondo y tamaño de fuente llamando al método 'actualizar':

ngOnChanges(changes: SimpleChanges) {

this.actualizar();

}

* Para probar la directiva debemos importar la directiva en nuestro componente 'app.component.ts':
* import { Component } from '@angular/core';
* import { RouterOutlet } from '@angular/router';
* import { ResaltadoDirective } from './resaltado.directive';
* @Component({
* selector: 'app-root',
* imports: [RouterOutlet, ResaltadoDirective],
* templateUrl: './app.component.html',
* styleUrls: ['./app.component.css']
* })
* export class AppComponent {
* colorselect = "green";
* tamanoFuente = 30;
* cambiarColor(col:string) {
* this.colorselect = col;
* }
* agrandar() {
* this.tamanoFuente++;
* }
* achicar() {
* this.tamanoFuente--;
* }
* }

En la clase definimos 2 atributos llamados colorselect y tamanoFuente que almacenan los valores que se le pasan a las propiedades de la directiva que hemos creado.

También modificamos el archivo 'app.component.html':

<p>Texto con resaltado : <span [appResaltado]="colorselect" [tamano]="tamanoFuente">prueba de directiva</span></p>

<button (click)="cambiarColor('red')">Rojo</button>

<button (click)="cambiarColor('yellow')">Amarillo</button>

<button (click)="agrandar()">Agrandar</button>

<button (click)="achicar()">Achicar</button>

<router-outlet />

Para inicializar la propiedad que define el color inicial de la directiva le asignamos:

[appResaltado]="colorselect"

En la misma etiqueta 'span' definimos el otro atributo:

[tamano]="tamanoFuente"

Si se presiona el botón con la etiqueta 'Rojo' se ejecuta el método:

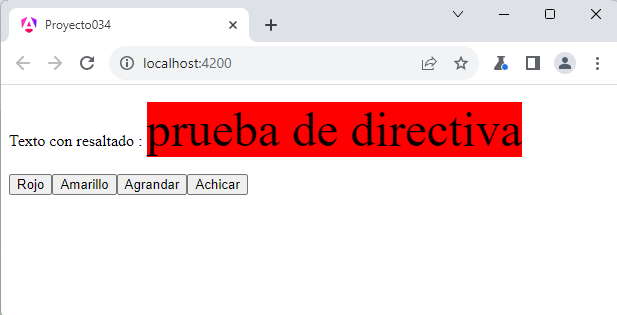
cambiarColor(col:string) {

this.colorselect = col;

}

Cuando cambiamos el valor del atributo 'colorselect', se actualiza el color de la etiqueta HTML que tiene asociada la directiva de atributo.

Si ejecutamos la aplicación tenemos como resultado:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto034/).

50 - Directivas de atributo - responder a eventos del usuario dentro de la directiva

Cuando creamos una directiva personalizada podemos capturar eventos dentro de la misma para reaccionar. Podemos por ejemplo detectar y reaccionar cuando el mouse entra o sale del elemento HTML que estamos aplicando la directiva.

Problema

Crear una directiva personalizada que se pueda asociar a cualquier elemento HTML y cuyo objetivo sea mediante la síntesis de voz hacer la lectura del contenido de texto que contiene la etiqueta. La lectura debe comenzar cuando el usuario dispone el mouse sobre la etiqueta y terminarla si saca la flecha del mouse.

* Crearemos primero el proyecto
* ng new proyecto035
* Procedemos a crear la directiva de atributo personalizada llamando a la misma 'textovoz' (indicamos que no queremos prefijo):
* ng generate directive textovoz --prefix

Se crean dos archivos.

* Procedemos a modificar el archivo 'textovoz.directive.ts' implementando la lógica de nuestra directiva:
* import { Directive, ElementRef, HostListener } from '@angular/core';
* @Directive({
* selector: '[textovoz]'
* })
* export class TextovozDirective {
* constructor(private elemento: ElementRef) {
* }
* @HostListener('mouseenter') entradaMouse() {
* speechSynthesis.speak(new SpeechSynthesisUtterance(this.elemento.nativeElement.textContent));
* }
* @HostListener('mouseleave') salidaMouse() {
* speechSynthesis.cancel();
* }
* }

Se inyecta al constructor un objeto de la clase 'ElementRef':

constructor(private elemento: ElementRef) {

Debemos importar 3 clases, que se almacenan en '@angular/core':

import { Directive, ElementRef, HostListener } from '@angular/core';

Para captura de eventos debemos utilizar la función decoradora @HostListener pasando como parámetro el nombre del evento a capturar:

@HostListener('mouseenter')

Cuando el usuario ingresa la flecha del mouse dentro de la componente, mediante el objeto 'speechSynthesis' y la clase 'SpeechSynthesisUtterance' procede la alocución del string que le pasamos como parámetro, el mismo lo rescatamos del contenido de la etiqueta HTML:

@HostListener('mouseenter') entradaMouse() {

speechSynthesis.speak(new SpeechSynthesisUtterance(this.elemento.nativeElement.textContent));

}

También capturamos el evento 'mouseleave' donde procedemos a cancelar la alocución del mensaje, en el caso que no se haya terminado previamente:

@HostListener('mouseleave') salidaMouse() {

speechSynthesis.cancel();

}

* Para probar la directiva 'textovoz' vamos a utilizar la componente que Angular nos ha creado por defecto, modificamos el archivo 'app.component.ts':
* import { Component } from '@angular/core';
* import { RouterOutlet } from '@angular/router';
* import { TextovozDirective } from './textovoz.directive';
* @Component({
* selector: 'app-root',
* imports: [RouterOutlet, TextovozDirective],
* templateUrl: './app.component.html',
* styleUrls: ['./app.component.css']
* })
* export class AppComponent {
* title = 'proyecto035';
* }

Y el archivo 'app.component.html':

<p>Como se pronuncia la palabra: <span textovoz>casa</span></p>

<p>Como se pronuncia la palabra: <span textovoz>ventana</span></p>

<p>Como se pronuncia la oración:</p>

<pre textovoz>

Aquí me pongo a cantar

al compás de la vihuela

que el hombre que lo desvela

una pena extraordinaria

como el ave solitaria

con el cantar se consuela

</pre>

<router-outlet />

Como vemos simplemente agregamos la directiva de atributo que hemos creado:

<span textovoz>casa</span>

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto035/).

51 - TypeScript

Hemos avanzado sobre muchas características del framework Angular y no nos hemos detenido sobre el lenguaje que lo soporta que es TypeScript.

TypeScript es un superconjunto de JavaScript que compila a JavaScript.

TypeScript agrega muchas características a JavaScript que nos facilitan la implementación de las aplicaciones.

Cada vez que compilamos la aplicación Angular:

C:\angulardevya\proyecto035> ng serve -o

Se convierten todos los archivos TypeScript (\*.ts) en archivos JavaScript (\*.js). Este proceso se hace en forma automática gracias a la herramienta Angular CLI. Como programadores en Angular solo nos concentramos en el desarrollo de la aplicación utilizando TypeScript y dejamos que la herramienta se encargue de todas las transformaciones para que luego se ejecute dentro de un navegador web (recordemos que los navegadores solo pueden interpretar archivos JavaScript y no TypeScript)

Cuando creamos un proyecto con Angular CLI se instala en la carpeta 'node\_modules' el compilador en la subcarpeta 'typescript'.

Si abrimos el archivo package.json podemos identificar en las dependencias de desarrollo la versión de TypeScript que requiere nuestro proyecto:

{

"name": "proyecto035",

"version": "0.0.0",

"scripts": {

"ng": "ng",

"start": "ng serve",

"build": "ng build",

"watch": "ng build --watch --configuration development",

"test": "ng test"

},

"private": true,

"dependencies": {

"@angular/animations": "^19.0.0",

"@angular/common": "^19.0.0",

"@angular/compiler": "^19.0.0",

"@angular/core": "^19.0.0",

"@angular/forms": "^19.0.0",

"@angular/platform-browser": "^19.0.0",

"@angular/platform-browser-dynamic": "^19.0.0",

"@angular/router": "^19.0.0",

"rxjs": "~7.8.0",

"tslib": "^2.3.0",

"zone.js": "~0.15.0"

},

"devDependencies": {

"@angular-devkit/build-angular": "^19.0.2",

"@angular/cli": "^19.0.2",

"@angular/compiler-cli": "^19.0.0",

"@types/jasmine": "~5.1.0",

"jasmine-core": "~5.4.0",

"karma": "~6.4.0",

"karma-chrome-launcher": "~3.2.0",

"karma-coverage": "~2.2.0",

"karma-jasmine": "~5.1.0",

"karma-jasmine-html-reporter": "~2.1.0",

**"typescript": "~5.6.2"**

}

}

Ventajas

* Facilita el desarrollo de aplicaciones complejas.
* Agrega la posibilidad de definir tipos (string, number, boolean etc.) a las variables que creamos. Con esta característica nos permite identificar problemas en nuestras aplicaciones en tiempo de compilación. Esta característica ayuda que nuestro programa sea más legible.  
  El origen del nombre de este lenguaje "TypeScript" es por la posibilidad de definir tipos (Type) a las variables.
* Definición de sintaxis más claras en la declaración de clases y sus propiedades.
* Definición de herencia de clases e implementación de interfaces.
* Definición propiedades privadas, protegidas y públicas.
* Captura de errores cuando se compila el programa en TypeScript.

TypeScript es un lenguaje creado por Microsoft y de distribución libre, el sitio oficial del lenguaje donde podemos consultar su documentación más actual está [aquí](https://www.typescriptlang.org/). De todos modos recordemos que no necesitamos descargarlo e instalarlo ya que Angular CLI siempre hace esto por nosotros cada vez que creamos un proyecto.

52 - TypeScript: tipado estático

Los lenguajes con tipado estático permiten detectar errores de asignación en tiempo de desarrollo.

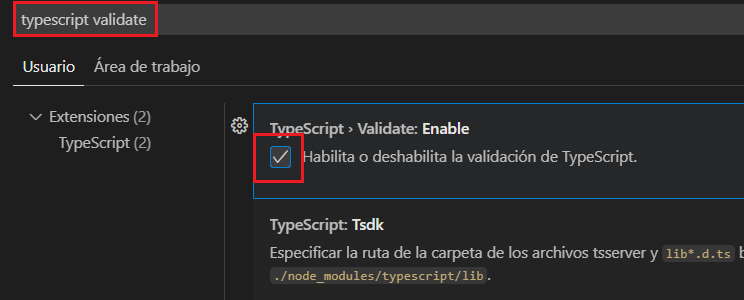
Por ejemplo si tenemos la definición de una variable 'number' y luego queremos asignarle un 'string' la herramienta de desarrollo la puede detectar (también si lo compilamos nos detecta un error)

Para probar los conceptos de TypeScript podemos crear un proyecto:

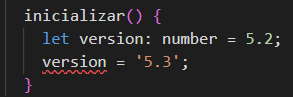
ng new proyecto036

JavaScript es un lenguaje de programación que usa un tipado dinámico, luego la comprobación de tipificación se realiza durante su ejecución en vez de durante la compilación.

Para activar en VSCode la verificación de tipos en TypeScript debemos entrar a la opción de "Configuración" y buscar "typescript validate" y proceder a activarlo:



Luego podemos probar en nuestro proyecto definir una variable de tipo number y posteriormente tratar de asignarle un string:



Tipo de datos básicos soportados por TypeScript.

* number

Permite almacenar tanto valores enteros como reales.

let edad: number = 23;

let altura: number = 1.92;

Si inicializamos la variable inmediatamente podemos dejar que TypeScript infiera el tipo de dato, es decir es lo mismo que escribir:

let edad = 23;

let altura = 1.92;

* string

Permite almacenar una cadena de caracteres:

let estudios:string = 'primarios';

Podemos almacenar entre las comillas cualquier caracter.

* boolean

Podemos almacenar el valor true o false:

let activo: boolean = true;

* Arreglos con elementos del mismo tipo

Podemos almacenar un conjunto de elementos del mismo tipo mediante vectores o arreglos:

let vector: number[] = [1, 4, 2];

vector.push(33);

for(let elemento of vector)

console.log(elemento);

Indicamos luego del tipo de dato los corchetes abiertos y cerrados:

let vector: number[]

Si queremos definir e inicializar algunas componentes del arreglo luego la sintaxis es:

let vector: number[] = [1, 4, 2];

El compilador de TypeScript nos podrá avisar del error si queremos agregar componentes que no sean de tipo 'number':

vector.push('juan'); //error

Otra sintaxis para definir un arreglo en TypeScript es mediante genéricos utilizando la clase Array:

let vector: Array<number> = [1, 4, 2];

vector.push(33);

for(let elemento of vector)

console.log(elemento);

Podemos probar dentro de una aplicación Angular los códigos anteriores implementando el método ngOnInit que pertenece a la interfaz OnInit.

La interfaz OnInit forma parte del ciclo de vida del componente de Angular. Se ejecuta después de que Angular ha creado el componente y ha configurado sus propiedades de entrada, pero antes de que se presente en la vista.

Puede utilizar este método para realizar acciones específicas que deben llevarse a cabo justo después de que el componente se haya inicializado.

import { Component, OnInit } from '@angular/core';

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title = "Prueba TypeScript";

ngOnInit() {

let vector: number[] = [1, 4, 2];

vector.push(33);

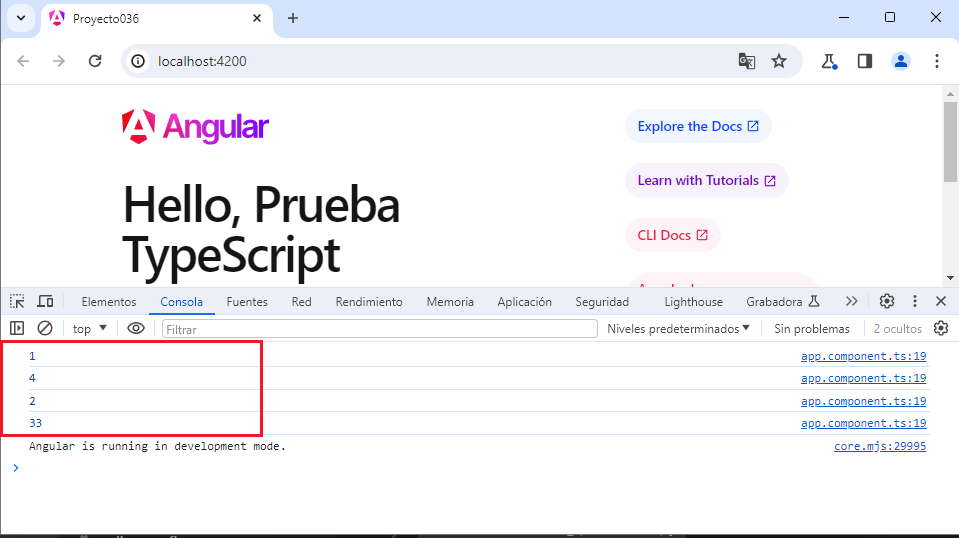
for (let elemento of vector)

console.log(elemento);

}

}

Luego si ejecutamos la aplicación y abrimos la consola del navegador podemos ver los valores almacenados en el vector:



* enum

El tipo de dato enum permite hacer nuestro código más legible:

import { Component, OnInit } from '@angular/core';

import { RouterOutlet } from '@angular/router';

enum Operacion {Suma, Resta, Multiplicacion, Division};

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title = "Prueba TypeScript";

ngOnInit() {

let operacion: Operacion = Operacion.Multiplicacion;

this.mostrarOperacion(operacion);

}

mostrarOperacion(operacion:Operacion) {

switch (operacion) {

case Operacion.Suma: {

console.log('Operación actual: Suma ');

break;

}

case Operacion.Resta: {

console.log('Operación actual: Resta ');

break;

}

case Operacion.Multiplicacion: {

console.log('Operación actual: Multiplicacion ');

break;

}

case Operacion.Division: {

console.log('Operación actual: Division ');

break;

}

}

}

}

Estamos declarando un nuevo tipo de dato que puede almacenar alguno de los cuatro valores indicados entre llaves:

enum Operacion {Suma, Resta, Multiplicacion, Division};

Luego podemos definir una variable de este tipo y almacenar uno de esos cuatro valores:

let operacion: Operacion = Operacion.Multiplicacion;

Cuando queremos saber que valor almacena la variable 'operacion' la comparamos por ejemplo con los valores posibles del tipo 'Operacion':

mostrarOperacion(operacion:Operacion) {

switch (operacion) {

case Operacion.Suma: {

console.log('Operación actual: Suma ');

break;

}

case Operacion.Resta: {

console.log('Operación actual: Resta ');

break;

}

case Operacion.Multiplicacion: {

console.log('Operación actual: Multiplicacion ');

break;

}

case Operacion.Division: {

console.log('Operación actual: Division ');

break;

}

}

}

Es una forma más elegante que tratar de resolver el problema con una variable de tipo 'number':

import { Component, OnInit } from '@angular/core';

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

title = "Prueba TypeScript";

ngOnInit() {

let operacion = 2; // significado 0 = suma / 1 = resta / 2 = multiplicacion / 3 = division

this.mostrarOperacion(operacion);

}

mostrarOperacion(operacion: number) {

switch (operacion) {

case 0: {

console.log('Operación actual: Suma ');

break;

}

case 1: {

console.log('Operación actual: Resta ');

break;

}

case 2: {

console.log('Operación actual: Multiplicacion ');

break;

}

case 3: {

console.log('Operación actual: Division ');

break;

}

}

}

}

* any

En las situaciones que debemos almacenar un dato en una variable y no sabemos de antemano que tipo se trata, TypeScript incorpora el tipo 'any'.

Es un tipo de dato que debemos evitarlo en lo posible ya que el compilador no nos puede ayudar y debemos esperar a ejecutar el programa para identificar errores.

Cuando utilizamos librerías heredadas de JavaScript es muy posible que nos veamos obligados a utilizar este tipo de dato.

let dato: any;

dato = 10;

console.log(dato);

dato = 'Hola';

console.log(dato);

dato = true;

console.log(dato);

dato = [1,2,3];

console.log(dato);

La variable dato muta su tipo a lo largo de la ejecución del algoritmo, primero es number, luego string, boolean y finalmente de tipo Array.

Podemos definir un Array con elementos de distinto tipo:

let vec: any[] = [10, 'Inicio', true];

Parámetros de métodos.

También cuando definimos funciones o métodos debemos definir el tipo de dato de los mismos:

mayor(valor1: number, valor2: number): number {

if (valor1 > valor2)

return valor1;

else

return valor2;

}

El método mayor recibe dos parámetros de tipo 'number' y retorna un tipo de dato 'number' que lo indicamos luego de los dos puntos.

Si no retorna dato un método podemos utilizar el tipo 'void':

export class AppComponent {

constructor() {

this.mostrarMensaje('Hola mundo');

}

mostrarMensaje(mensaje: string): void {

alert(mensaje);

}

}

Variable de tipo unión.

TypeScript permite definir variables que almacenen dos o más tipos de datos. Puede almacenar un único valor en un determinado momento, pero dicho valor puede variar entre los tipos indicados en la definición:

let edad: number | string;

edad=34;

console.log(edad);

edad='20 años';

console.log(edad);

Mediante el caracter | separamos los tipos posibles de datos que puede almacenar la variable. La variable 'edad' puede almacenar un valor de tipo 'number' o 'string':

let edad: number | string;

53 - TypeScript: clases

TypeScript incorpora muchas características de la programación orientada a objetos disponibles en lenguajes como Java y C#.

La sintaxis básica de una clase puede ser:

class Persona {

nombre: string;

edad: number;

constructor(nombre:string, edad:number) {

this.nombre = nombre;

this.edad = edad;

}

imprimir() {

console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);

}

}

let persona1: Persona;

persona1 = new Persona('Juan', 45);

persona1.imprimir();

Los atributos se definen fuera de los métodos. Para acceder a los mismos dentro de los métodos debemos anteceder la palabra clave 'this':

imprimir() {

console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);

}

El constructor es el primer método que se ejecuta en forma automática al crear un objeto de la clase 'Persona':

constructor(nombre:string, edad:number) {

this.nombre = nombre;

this.edad = edad;

}

Podemos probar este código agregándolo previo a la componente y luego ver la consola del navegador:

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

import { RouterOutlet } from '@angular/router';

class Persona {

nombre: string;

edad: number;

constructor(nombre: string, edad: number) {

this.nombre = nombre;

this.edad = edad;

}

imprimir() {

console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);

}

}

let persona1: Persona;

persona1 = new Persona('Juan', 45);

persona1.imprimir();

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

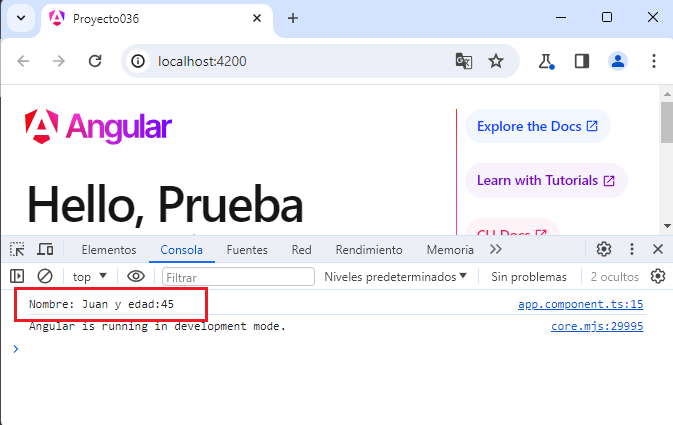
})

export class AppComponent {

title = "Prueba TypeScript";

}

En la consola comprobamos los valores:



Modificadores de acceso a propiedades y métodos.

Podemos definir propiedades y métodos privados y públicos antecediendo las palabras claves 'private' y 'public'.

Veamos un ejemplo:

class Dado {

private valor: number = 1;

public tirar() {

this.generar();

}

private generar() {

this.valor = Math.floor(Math.random() \* 6) + 1;

}

public imprimir() {

console.log(`Salió el valor ${this.valor}`);

}

}

let dado1 = new Dado();

dado1.tirar();

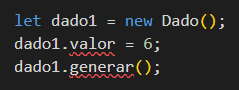
dado1.imprimir();

El objeto 'dado1' tiene acceso a todos los atributos y métodos que tienen el modificador 'public' por eso podemos llamar a los métodos 'tirar' e 'imprimir':

dado1.tirar();

dado1.imprimir();

Se generará un error si queremos acceder al atributo 'valor' o al método 'generar':



Si no agregamos modificador de acceso por defecto es 'public', luego tendremos el mismo resultado si utilizamos la siguiente sintaxis para declarar la clase:

class Dado {

private valor: number = 1;

tirar() {

this.generar();

}

private generar() {

this.valor = Math.floor(Math.random() \* 6) + 1;

}

imprimir() {

console.log(`Salió el valor ${this.valor}`);

}

}

let dado1=new Dado();

dado1.tirar();

dado1.imprimir();

Es decir nos ahorramos de escribir 'public' antes de las propiedades y métodos que queremos definir con esta característica.

Todo atributo de una clase debe inicializarse cuando se lo define o en el constructor, sino se genera un error.

Si por algún motivo no queremos inicializar el atributo, debemos utilizar el modificador "!" (Non-null assertion operator)

Podemos indicar a TypeScript que confíe en que la propiedad no será nula utilizando el operador de afirmación de no nulo (!). Sin embargo, debes estar seguro de que nunca intentarás acceder a la propiedad antes de que se le haya asignado un valor:

class Dado {

private valor!: number;

public tirar() {

this.generar();

}

private generar() {

this.valor = Math.floor(Math.random() \* 6) + 1;

}

public imprimir() {

console.log(`Salió el valor ${this.valor}`);

}

}

let dado1 = new Dado();

dado1.tirar()

dado1.imprimir()

Luego si nos olvidamos de llamar al método 'tirar' y llamamos al método imprimir, se mostrará el valor 'undefined' que es el contenido en el atributo 'valor'.

Definición e inicialización de propiedades en los parámetros del constructor.

Esta característica no es común en otros lenguajes orientados a objetos y tiene por objetivo crear clases más breves.

En TypeScript podemos definir algunas propiedades de la clase en la zona de parámetros del constructor, con esto nos evitamos de su declaración fuera de los métodos.

Por ejemplo la clase Persona la podemos codificar con ésta otra sintaxis:

class Persona {

constructor(public nombre:string, public edad:number) { }

imprimir() {

console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);

}

}

let persona1: Persona;

persona1 = new Persona('Juan', 45);

persona1.imprimir();

Como vemos el constructor tiene un bloque de llaves vacías ya que no tenemos que implementar ningún código en su interior, pero al anteceder el modificador de acceso en la zona de parámetros los mismos pasan a ser propiedades de la clase y no parámetros:

constructor(public nombre:string, public edad:number) { }

Podemos sin problemas definir propiedades tanto 'public' como 'private'.

La definición de propiedades en la zona de parámetros solo se puede hacer en el constructor de la clase y no está permitido en cualquier otro método.

Modificador readonly

Disponemos además de los modificadores 'private' y 'public' uno llamado 'readonly'. Mediante este modificador el valor de la propiedad solo puede ser cargado en el constructor o al momento de definirlo y luego no puede ser modificado ni desde un método de la clase o fuera de la clase.

Veamos un ejemplo con una propiedad 'readonly':

class Articulo {

readonly codigo: number;

descripcion: string;

precio: number;

constructor(codigo:number, descripcion:string, precio:number) {

this.codigo=codigo;

this.descripcion=descripcion;

this.precio=precio;

}

imprimir() {

console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);

}

}

let articulo1: Articulo;

articulo1 = new Articulo(1,'papas',12.5);

articulo1.imprimir();

Una vez que se inicia la propiedad 'codigo' en el constructor su valor no puede cambiar:

imprimir() {

this.codigo=7; //Error

console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);

}

El mismo error se produce si tratamos de cambiar su valor desde fuera de la clase:

let articulo1: Articulo;

articulo1 = new Articulo(1,'papas',12.5);

articulo1.codigo=7; //Error

articulo1.imprimir();

Podemos utilizar también la sintaxis abreviada de propiedades:

class Articulo {

constructor(readonly codigo:number, public descripcion:string, public precio:number) { }

imprimir() {

console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);

}

}

Propiedades estáticas

Las propiedades estáticas pertenecen a la clase y no a las instancias de la clase. Se las define antecediendo el modificador 'static'.

Con un ejemplo quedará claro este tipo de propiedades:

class Dado {

private valor: number = 1;

static tiradas:number=0;

tirar() {

this.generar();

}

private generar() {

this.valor = Math.floor(Math.random() \* 6) + 1 ;

Dado.tiradas++;

}

imprimir() {

console.log(`Salió el valor ${this.valor}`);

}

}

let dado1=new Dado();

dado1.tirar();

dado1.imprimir();

let dado2=new Dado();

dado2.tirar();

dado2.imprimir();

console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2

Una propiedad estática requiere el modificador 'static' previo a su nombre:

static tiradas:number=0;

Para acceder a dichas propiedades debemos anteceder el nombre de la clase y no la palabra clave 'this':

Dado.tiradas++;

No importan cuantos objetos de la clase se definan luego todos esos objetos comparten la misma variable estática:

let dado1=new Dado();

dado1.tirar();

dado1.imprimir();

let dado2=new Dado();

dado2.tirar();

dado2.imprimir();

console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2

Es por eso que la propiedad 'tiradas' almacena un 2 luego de tirar el primer y segundo dado.

La propiedad 'valor' es independiente en cada dado pero la propiedad 'tiradas' es compartida por los dos objetos.

Métodos estáticas

Igual que las propiedades estáticas los métodos estáticos se los accede por el nombre de la clase. Este tipo de métodos solo pueden acceder a propiedades estáticas.

class Matematica {

static mayor(v1:number, v2: number): number {

if (v1>v2)

return v1;

else

return v2;

}

static menor(v1:number, v2: number): number {

if (v1<v2)

return v1;

else

return v2;

}

static aleatorio(inicio: number, fin: number): number {

return Math.floor((Math.random()\*(fin+1-inicio))+inicio);

}

}

let x1=Matematica.aleatorio(1,10);

let x2=Matematica.aleatorio(1,10);

console.log(`El mayor entre ${x1} y ${x2} es ${Matematica.mayor(x1,x2)}   
`);

console.log(`El menor entre ${x1} y ${x2} es ${Matematica.menor(x1,x2)}`);

Debemos anteceder la palabra clave static al nombre del método.

Cuando llamamos a un método debemos anteceder también el nombre de la clase, no hace falta definir una instancia u objeto de la clase:

let x1=Matematica.aleatorio(1,10);

54 - TypeScript: funciones y métodos

TypeScript aporta varias características que JavaScript no dispone hasta el momento cuando tenemos que plantear funciones y métodos.

Parámetros tipados y funciones que retornan un valor.

Podemos indicar a cada parámetro el tipo de dato que puede recibir y también el tipo de dato que retorna la función o método en caso que estemos en una clase:

function sumar(valor1:number, valor2:number): number {

return valor1+valor2;

}

console.log(sumar(10, 5));

La función sumar recibe dos parámetros de tipo number y retorna un valor de tipo number. Luego si llamamos a esta función enviando un valor distinto a number el compilador nos avisará del error:

console.log(sumar('juan', 'carlos'));

Se genera un error: Argument of type 'string' is not assignable to parameter of type 'number'.

Inclusive editores de texto moderno como Visual Studio Code pueden antes de compilarse avisar del error.

El tipado estático favorece a identificar este tipo de errores antes de ejecutar la aplicación. Lo mismo cuando una función retorna un dato debemos indicar al final de la misma dicho tipo:

function sumar(valor1:number, valor2:number): **number** {

La función sumar retorna un valor de tipo number.

Luego si la función retorna un tipo distinto a number se genera un error:

function sumar(valor1:number, valor2:number): number {

return 'Hola mundo';

}

Como estamos retornando un string se genera el error: Type 'string' is not assignable to type 'number'

Funciones anónimas.

Una función anónima no especifica un nombre. Son semejantes a JavaScript con la salvedad de la definición de tipos para los parámetros:

const funcSumar = function (valor1:number, valor2:number): number {

return valor1 + valor2;

}

console.log(funcSumar(4, 9));

Parámetros opcionales.

En TypeScript debemos agregar el caracter '?' al nombre del parámetro para indicar que el mismo puede o no llegar un dato:

function sumar(valor1: number, valor2: number, valor3?: number): number {

if (valor3)

return valor1 + valor2 + valor3;

else

return valor1 + valor2;

}

console.log(sumar(5, 4));

console.log(sumar(5, 4, 3));

El tercer parámetro es opcional:

function sumar(valor1: number, valor2: number, **valor3?: number**): number {

Luego a la función 'sumar' la podemos llamar pasando 2 o 3 valores numéricos:

console.log(sumar(5,4));

console.log(sumar(5,4,3));

Si pasamos una cantidad de parámetros distinta a 2 o 3 se genera un error en tiempo de compilación: ' error TS2554: Expected 2-3 arguments, but got 4.'

Los parámetros opcionales deben ser los últimos parámetros definidos en la función. Puede tener tantos parámetros opcionales como se necesiten.

Parámetros por defecto.

Esta característica de TypeScript nos permite asignar un valor por defecto a un parámetro para los casos en que la llamada a la misma no se le envíe.

function sumar(valor1: number, valor2: number, valor3: number = 0): number {

return valor1 + valor2 + valor3;

}

console.log(sumar(5, 4));

console.log(sumar(5, 4, 3));

El tercer parámetro almacena un cero si no se lo pasamos en la llamada:

console.log(sumar(5, 4));

Puede haber varios valores por defecto, pero deben ser los últimos. Es decir primero indicamos los parámetros que reciben datos en forma obligatoria cuando los llamamos y finalmente indicamos aquellos que tienen valores por defecto.

parámetros Rest.

Otra característica de TypeScript es la posibilidad de pasar una lista indefinida de valores y que los reciba un vector.

El concepto de parámetro Rest se logra antecediendo tres puntos al nombre del parámetro:

function sumar(...valores: number[]) {

let suma = 0;

for (let x = 0; x < valores.length; x++)

suma += valores[x];

return suma;

}

console.log(sumar(10, 2, 44, 3));

console.log(sumar(1, 2));

console.log(sumar());

El parámetro 'valores' se le anteceden los tres puntos seguidos e indicamos que se trata de un vector de tipo 'number'. Cuando llamamos a la función le pasamos una lista de valores enteros que luego la función los empaqueta en el vector:

console.log(sumar(10, 2, 44, 3));

console.log(sumar(1, 2));

console.log(sumar());

La función con un parámetro Rest puede tener otros parámetros pero se deben declarar antes.

Los parámetros Rest no pueden tener valores por defecto.

operador Spread.

El operador Spread permite descomponer una estructura de datos en elementos individuales. Es la operación inversa de los parámetros Rest. La sintaxis se aplica anteponiendo al nombre de la variable tres puntos:

function sumar(valor1: number, valor2: number, valor3: number): number {

return valor1 + valor2 + valor3;

}

const vec: [number, number, number] = [10, 20, 30];

const s = sumar(...vec);

console.log(s);

Otra cosa importante es hacer notar que el arreglo indicamos el tipo para cada elemento.

Funciones callbacks

Una función callback es una función que se pasa a otra función como parámetro y dentro de la misma es llamada.

function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number): number {

return func(valor1, valor2);

}

console.log(operar(3, 7, (x: number,y: number): number => {

return x+y;

}))

console.log(operar(3, 7, (x: number,y: number): number => {

return x-y;

}))

console.log(operar(3, 7, (x: number,y: number): number => {

return x\*y;

}))

La función operar recibe tres parámetros, los dos primeros son de tipo 'number' y el tercero es de tipo función:

function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number): number {

La función que debe recibir debe tener como parámetros dos 'number' y retornar un 'number'.

Cuando llamamos a la función además de los dos enteros le debemos pasar una función que reciba dos 'number' y retorne un 'number':

console.log(operar(3, 7, (x: number,y: number): number => {

return x+y;

}))

Como podemos observar llamamos a la función 'operar' tres veces y le pasamos funciones que procesan los dos enteros para obtener su suma, resta y multiplicación.

Podemos llamar a la función 'operar' con una sintaxis más concisa como en JavaScript:

console.log(operar(3, 7, (x: number,y: number): number => x+y))

console.log(operar(3, 7, (x: number,y: number): number => x-y))

console.log(operar(3, 7, (x: number,y: number): number => x\*y))

Para hacer más claro nuestro código TypeScript mediante la palabra clave type permite crear nuevos tipos y luego reutilizarlos:

type Operacion = (x: number, y: number) => number;

function operar(valor1: number, valor2: number, func: Operacion): number {

return func(valor1, valor2);

}

console.log(operar(3, 7, (x: number, y: number): number => x + y))

console.log(operar(3, 7, (x: number, y: number): number => x - y))

console.log(operar(3, 7, (x: number, y: number): number => x \* y))

El tipo Operacion tiene la firma de una función con dos parámetros de tipo 'number' y el retorno de un 'number'. Luego cuando declaramos la función operar definimos el tercer parámetro llamado 'func' de tipo 'Operacion':

function operar(valor1: number, valor2: number, func: Operacion): number {

Parámetros de tipo unión.

Vimos en otro concepto que podemos definir variables que pueden almacenar más de un tipo de dato indicando los mismos el operador '|':

let edad: number | string;

edad=34;

console.log(edad);

edad='20 años';

console.log(edad);

Con parámetros podemos utilizar la misma sintaxis:

function sumar(valor1: number | string, valor2: number | string ): number | string {

if (typeof valor1 ==='number' && typeof valor2 ==='number')

return valor1+valor2;

else

return valor1.toString() + valor2.toString();

}

console.log(sumar(4, 5));

console.log(sumar('Hola ', 2));

console.log(sumar('Hola ', 'Mundo'));

En este tipo de caso deberemos identificar que operación realizar según los tipos de datos de los parámetros. En el ejemplo si los dos parámetros se reciben tipos de datos 'number' procedemos a sumarlos como enteros:

if (typeof valor1 ==='number' && typeof valor2 ==='number')

return valor1+valor2;

En el caso contrario con que uno de los dos valores sea de tipo 'string' procedemos a concatenarlos, previamente los convertimos a string:

else

return valor1.toString() + valor2.toString();

Acotaciones

Hemos hecho siempre ejemplos con funciones, pero todos estos conceptos se aplican si planteamos métodos dentro de una clase:

class Operacion {

sumar(...valores:number[]) {

let suma=0;

for(let x=0;x<valores.length;x++)

suma+=valores[x];

return suma;

}

}

let op: Operacion;

op=new Operacion();

console.log(op.sumar(1,2,3));

55 - TypeScript: herencia

La herencia es otra característica fundamental de la programación orientada a objetos y TypeScript lo implementa.

La herencia significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todas los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otros atributos y métodos propios.

Veamos con un ejemplo la sintaxis que plantea TypeScript para implementar la herencia:

class Persona {

protected nombre: string;

protected edad: number;

constructor(nombre: string, edad: number) {

this.nombre = nombre;

this.edad = edad;

}

imprimir() {

console.log(`Nombre: ${this.nombre}`);

console.log(`Edad: ${this.edad}`);

}

}

class Empleado extends Persona {

private sueldo: number;

constructor(nombre: string, edad: number, sueldo: number) {

super(nombre, edad);

this.sueldo = sueldo;

}

override imprimir() {

super.imprimir();

console.log(`Sueldo: ${this.sueldo}`);

}

pagaImpuestos() {

if (this.sueldo > 5000)

console.log(`${this.nombre} debe pagar impuestos`);

else

console.log(`${this.nombre} no debe pagar impuestos`);

}

}

El resultado de ejecutar en la consola del navegador es (recordar que la consola la podemos abrir desde el navegador presionando la tecla F12 ):



Mediante la palabra clave extends indicamos el nombre de la clase padre. Una clase puede heredar de una sola clase (en este ejemplo 'Persona'):

class Empleado extends Persona {

private sueldo: number;

constructor(nombre: string, edad: number, sueldo: number) {

super(nombre, edad);

this.sueldo = sueldo;

}

La subclase Empleado puede acceder a las propiedades de la clase padre si los mismos se definieron en forma public o protected, debemos agregar la palabra clave 'override' para sobreescribir un método de la clase padre:

override imprimir() {

super.imprimir();

console.log(`Sueldo: ${this.sueldo}`);

}

Con el modificador protected permitimos que la subclase pueda acceder a los atributos de la clase padre pero luego donde definamos un objeto de esta clase no los pueda acceder y permanezcan encapsulados:

const empleado1=new Empleado('Ana', 22, 7000);

empleado1.nombre='facundo'; //error

Clases abstractas

En algunas situaciones tenemos métodos y propiedades comunes a un conjunto de clases, podemos agrupar dichos métodos y propiedades en una clase abstracta.

Hay una sintaxis especial en TypeScript para indicar que una clase es abstracta.

No se pueden definir objetos de una clase abstracta y seguramente será heredada por otras clases de las que si podremos definir objetos.

Problema: Declarar una clase abstracta que represente una Operación. Definir en la misma tres propiedades valor1, valor2 y resultado, y tres métodos: constructor, imprimir y operar (éste último hacerlo abstracto). Plantear dos clases llamadas Suma y Resta que hereden de la clase Operación e implementen el método abstracto operar.

abstract class Operacion {

public valor1: number;

public valor2: number;

public resultado: number = 0;

constructor(v1: number, v2: number) {

this.valor1 = v1;

this.valor2 = v2;

}

abstract operar(): void;

imprimir() {

console.log(`Resultado: ${this.resultado}`);

}

}

class Suma extends Operacion {

constructor(v1: number, v2: number) {

super(v1, v2);

}

operar() {

this.resultado = this.valor1 + this.valor2;

}

}

class Resta extends Operacion {

constructor(v1: number, v2: number) {

super(v1, v2);

}

operar() {

this.resultado = this.valor1 - this.valor2;

}

}

let suma1: Suma;

suma1 = new Suma(10, 4);

suma1.operar();

suma1.imprimir();

let resta1: Resta;

resta1 = new Resta(10, 4);

resta1.operar();

resta1.imprimir();

Mediante la palabra clave abstract indicamos que la clase debe definirse como abstracta, luego no se pueden definir objetos de la clase Operacion:

abstract class Operacion {

public valor1: number;

public valor2: number;

public resultado: number = 0;

constructor(v1: number, v2: number) {

this.valor1 = v1;

this.valor2 = v2;

}

abstract operar(): void;

imprimir() {

console.log(`Resultado: ${this.resultado}`);

}

}

Dentro de la clase abstracta definimos un método abstracto llamado operar, esto obliga a todas las clases que heredan de 'Operacion' implementar el algoritmo de dicho método, sino se genera un error en tiempo de compilación.

La subclase Suma al heredar de Operación implementa el método operar:

class Suma extends Operacion {

constructor(v1: number, v2: number) {

super(v1, v2);

}

operar() {

this.resultado = this.valor1 + this.valor2;

}

}

Solo podemos definir objetos de las clases Suma y Resta. Se genera un error si tratamos de crear un objeto de la clase Operacion:

let suma1: Suma;

suma1=new Suma(10, 4);

suma1.operar();

suma1.imprimir();

56 - TypeScript: interfaces

Una interface declara una serie de métodos y propiedades que deben ser implementados luego por una o más clases.

Las interfaces vienen a suplir la imposibilidad de herencia múltiple.

Por ejemplo podemos tener dos clases que representen un avión y un helicóptero. Luego plantear una interface con un método llamado volar. Las dos clases pueden implementar dicha interface y codificar el método volar (los algoritmos seguramente sean distintos pero el comportamiento de volar es común tanto a un avión como un helicóptero)

La sintaxis en TypeScript para declarar una interface es:

interface [nombre de la interface] {

[declaración de propiedades]

[declaración de métodos]

}

Problema

Definir una interface llamada Punto que declare un método llamado imprimir. Luego declarar dos clases que la implementen.

interface Punto {

imprimir(): void;

}

class PuntoPlano implements Punto {

constructor(private x: number, private y: number) { }

imprimir() {

console.log(`Punto en el plano: (${this.x},${this.y})`);

}

}

class PuntoEspacio implements Punto {

constructor(private x: number, private y: number, private z: number) { }

imprimir() {

console.log(`Punto en el espacio: (${this.x},${this.y},${this.z})`);

}

}

let puntoPlano1: PuntoPlano;

puntoPlano1 = new PuntoPlano(10, 4);

puntoPlano1.imprimir();

let puntoEspacio1: PuntoEspacio;

puntoEspacio1 = new PuntoEspacio(20, 50, 60);

puntoEspacio1.imprimir();

Para declarar una interface en TypeScript utilizamos la palabra clave interface y seguidamente su nombre. Luego entre llaves indicamos todas las cabeceras de métodos y propiedades. En nuestro ejemplo declaramos la interface Punto e indicamos que quien la implemente debe definir un método llamado imprimir sin parámetros y que no retorna nada:

interface Punto {

imprimir(): void;

}

Por otro lado declaramos dos clases llamados PuntoPlano con dos propiedades y PuntoEspacio con tres propiedades, además indicamos que dichas clases implementarán la interface Punto:

class PuntoPlano implements Punto {

constructor(private x: number, private y: number) { }

imprimir() {

console.log(`Punto en el plano: (${this.x},${this.y})`);

}

}

class PuntoEspacio implements Punto {

constructor(private x: number, private y: number, private z: number) { }

imprimir() {

console.log(`Punto en el espacio: (${this.x},${this.y},${this.z})`);

}

}

La sintaxis para indicar que una clase implementa una interface requiere disponer la palabra clave implements y en forma seguida el o los nombres de interfaces a implementar. Si una clase hereda de otra también puede implementar una o más interfaces.

El método imprimir en cada clase se implementa en forma distinta, en uno se imprimen 3 propiedades y en la otra se imprimen 2 propiedades.

Luego definimos un objeto de la clase PuntoPlano y otro de tipo PuntoEspacio:

let puntoPlano1: PuntoPlano;

puntoPlano1 = new PuntoPlano(10, 4);

puntoPlano1.imprimir();

let puntoEspacio1: PuntoEspacio;

puntoEspacio1 = new PuntoEspacio(20, 50, 60);

puntoEspacio1.imprimir();

Si una clase indica que implementa una interfaz y luego no se la codifica, se genera un error en tiempo de compilación informándonos de tal situación (inclusive el editor Visual Studio Code detecta dicho error antes de compilar):

prueba.ts(5,7): error TS2420: Class 'PuntoPlano' incorrectly implements interface 'Punto'.

Property 'imprimir' is missing in type 'PuntoPlano'.

prueba.ts(20,13): error TS2339: Property 'imprimir' does not exist on type 'PuntoPlano'.

Este error se produce si codificamos la clase sin implementar el método imprimir:

class PuntoPlano implements Punto{

constructor(private x:number, private y:number) {}

}

Problema

Se tiene la siguiente interface:

interface Figura {

superficie: number;

perimetro: number;

calcularSuperficie(): number;

calcularPerimetro(): number;

}

Declar dos clases que representen un Cuadrado y un Rectángulo. Implementar la interface Figura en ambas clases.

interface Figura {

superficie: number;

perimetro: number;

calcularSuperficie(): number;

calcularPerimetro(): number;

}

class Cuadrado implements Figura {

superficie: number;

perimetro: number;

constructor(private lado:number) {

this.superficie = this.calcularSuperficie();

this.perimetro = this.calcularPerimetro();

}

calcularSuperficie(): number {

return this.lado \* this.lado;

}

calcularPerimetro(): number {

return this.lado \* 4;

}

}

class Rectangulo implements Figura {

superficie: number;

perimetro: number;

constructor(private ladoMayor:number, private ladoMenor:number) {

this.superficie = this.calcularSuperficie();

this.perimetro = this.calcularPerimetro();

}

calcularSuperficie(): number {

return this.ladoMayor \* this.ladoMenor;

}

calcularPerimetro(): number {

return (this.ladoMayor \* 2) + (this.ladoMenor \* 2);

}

}

let cuadrado1: Cuadrado;

cuadrado1 = new Cuadrado(10);

console.log(`Perimetro del cuadrado : ${cuadrado1.calcularPerimetro()}`);

console.log(`Superficie del cuadrado : ${cuadrado1.calcularSuperficie()}`);

let rectangulo1: Rectangulo;

rectangulo1 = new Rectangulo(10, 5);

console.log(`Perimetro del rectangulo : ${rectangulo1.calcularPerimetro()}`);

console.log(`Superficie del rectangulo: ${rectangulo1.calcularSuperficie()}`);

En este problema la interface Figura tiene dos métodos que deben ser implementados por las clases y dos propiedades que también deben definirlos:

interface Figura {

superficie: number;

perimetro: number;

calcularSuperficie(): number;

calcularPerimetro(): number;

}

La clase Cuadrado indica que implementa la interface Figura, esto hace necesario que se implementen los métodos calcularSuperficie y calcularPerimetro, y las dos propiedades:

class Cuadrado implements Figura {

superficie: number;

perimetro: number;

constructor(private lado:number) {

this.superficie = this.calcularSuperficie();

this.perimetro = this.calcularPerimetro();

}

calcularSuperficie(): number {

return this.lado \* this.lado;

}

calcularPerimetro(): number {

return this.lado \* 4;

}

}

La clase Cuadrado tiene una propiedad llamada lado que la recibe el constructor.

De forma similar la clase Rectangulo implementa la interface Figura:

class Rectangulo implements Figura {

superficie: number;

perimetro: number;

constructor(private ladoMayor:number, private ladoMenor:number) {

this.superficie = this.calcularSuperficie();

this.perimetro = this.calcularPerimetro();

}

calcularSuperficie(): number {

return this.ladoMayor \* this.ladoMenor;

}

calcularPerimetro(): number {

return (this.ladoMayor \* 2) + (this.ladoMenor \* 2);

}

}

Finalmente definimos un objeto de la clase Cuadrado y otro de la clase Rectangulo, luego llamamos a los métodos calcularPerimetro y calcularSuperficie para cada objeto:

let cuadrado1: Cuadrado;

cuadrado1 = new Cuadrado(10);

console.log(`Perimetro del cuadrado : ${cuadrado1.calcularPerimetro()}`);

console.log(`Superficie del cuadrado : ${cuadrado1.calcularSuperficie()}`);

let rectangulo1: Rectangulo;

rectangulo1 = new Rectangulo(10, 5);

console.log(`Perimetro del rectangulo : ${rectangulo1.calcularPerimetro()}`);

console.log(`Superficie del cuadrado : ${rectangulo1.calcularSuperficie()}`);

Las interfaces exige que una clase siga las especificaciones de la misma y se implementen algoritmos más robustos. En nuestro ejemplo tanto la clase Rectangulo como Cuadrado tienen una forma similar de trabajar gracias a que implementan la interfaz Figura.

Parámetros de tipo interface.

Un método o función puede recibir como parámetro una interface. Luego le podemos pasar objetos de distintas clases que implementan dicha interface:

interface Figura {

superficie: number;

perimetro: number;

calcularSuperficie(): number;

calcularPerimetro(): number;

}

class Cuadrado implements Figura {

superficie: number;

perimetro: number;

constructor(private lado:number) {

this.superficie = this.calcularSuperficie();

this.perimetro = this.calcularPerimetro();

}

calcularSuperficie(): number {

return this.lado \* this.lado;

}

calcularPerimetro(): number {

return this.lado \* 4;

}

}

class Rectangulo implements Figura {

superficie: number;

perimetro: number;

constructor(private ladoMayor:number, private ladoMenor:number) {

this.superficie = this.calcularSuperficie();

this.perimetro = this.calcularPerimetro();

}

calcularSuperficie(): number {

return this.ladoMayor \* this.ladoMenor;

}

calcularPerimetro(): number {

return (this.ladoMayor \* 2) + (this.ladoMenor \* 2);

}

}

function imprimir(fig: Figura) {

console.log(`Perimetro: ${fig.calcularPerimetro()}`);

console.log(`Superficie: ${fig.calcularSuperficie()}`);

}

let cuadrado1: Cuadrado;

cuadrado1 = new Cuadrado(10);

console.log('Datos del cuadrado');

imprimir(cuadrado1);

let rectangulo1: Rectangulo;

rectangulo1 = new Rectangulo(10, 5);

console.log('Datos del rectángulo');

imprimir(rectangulo1);

La función imprimir recibe como parámetro fig que es de tipo Figura:

function imprimir(fig: Figura) {

console.log(`Perimetro: ${fig.calcularPerimetro()}`);

console.log(`Superficie: ${fig.calcularSuperficie()}`);

}

Podemos luego llamar a la función imprimir pasando tanto objetos de la clase Cuadrado como Rectangulo:

imprimir(cuadrado1);

imprimir(rectangulo1);

Es importante notar que solo podemos acceder a los métodos y propiedades definidos en la interfaz y no a propiedades y métodos propios de cada clase.

Creación de objetos a partir de una interface.

TypeScript permite crear objetos a partir de una interfaz. La sintaxis para dicha creación es:

interface Punto {

x: number;

y: number;

}

let punto1: Punto;

punto1 = {x:10, y:20};

console.log(punto1);

No podemos utilizar el operador new para la creación del objeto.

Podemos definir la variable e inmediatamente iniciarla:

let punto1: Punto = {x:10, y:20};

Propiedades opcionales.

Una interface puede definir propiedades opcionales que luego la clase que la implementa puede o no definirlas. Se utiliza la misma sintaxis de los parámetros opcionales, es decir se le agrega el caracter '?' al final del nombre de la propiedad.

interface Punto {

x: number;

y: number;

z?: number;

}

let puntoPlano: Punto = {x:10, y:20};

console.log(puntoPlano);

let puntoEspacio: Punto = {x:10, y:20, z:70};

console.log(puntoEspacio);

Como vemos el objeto 'puntoPlano' solo implementa las propiedades 'x' e 'y'.

Se produce un error en tiempo de compilación si no implementamos todas las propiedades obligatorias, por ejemplo:

let puntoPlano: Punto = {x:10};

Esta línea genera un error ya que solo se define la propiedad 'x' y falta definir la propiedad 'y'.

Herencia de interfaces.

TypeScript permite que una interface herede de otra:

interface Punto {

x: number;

y: number;

}

interface Punto3D extends Punto {

z: number;

}

let punto1: Punto = {x:10, y:20};

let punto2: Punto3D = {x:23, y:13, z:12};

console.log(punto1);

console.log(punto2);

57 - TypeScript: clases genéricas

TypeScript permite crear clases y funciones que administren distintos tipos de datos. Esta característica no existe en JavaScript

Se utilizan mucho para la administración de colecciones de datos (pilas, colas, listas etc.)

Para entender las ventajas de definir clases genéricas implementaremos los algoritmos para administrar una pila de enteros y una pila de string. Primero lo haremos utilizando clases tradicionales y luego mediante una clase genérica.

Para concentrarnos en la sintaxis plantearemos la pila utilizando un vector donde definiremos los dos métodos fundamentales de insertar y extraer (no haremos ningún tipo de validaciones por simplicidad)

class PilaEnteros {

private vec: number[] = [];

insertar(x: number) {

this.vec.push(x);

}

extraer() {

if (this.vec.length > 0)

return this.vec.pop();

else

return null;

}

}

class PilaStrings {

private vec: string[] = [];

insertar(x: string) {

this.vec.push(x);

}

extraer() {

if (this.vec.length > 0)

return this.vec.pop();

else

return null;

}

}

let pila1 = new PilaEnteros();

pila1.insertar(20);

pila1.insertar(43);

pila1.insertar(1);

console.log(pila1.extraer());

let pila2 = new PilaStrings();

pila2.insertar('juan');

pila2.insertar('ana');

pila2.insertar('luis');

console.log(pila2.extraer());

Como podemos analizar hemos planteado dos clases, una para administrar una pila con tipos de dato enteros:

class PilaEnteros {

private vec: number[] = [];

insertar(x: number) {

this.vec.push(x);

}

extraer() {

if (this.vec.length > 0)

return this.vec.pop();

else

return null;

}

}

Y por otro lado otra clase para administrar una pila de tipo de dato string:

class PilaStrings {

private vec: string[] = [];

insertar(x: string) {

this.vec.push(x);

}

extraer() {

if (this.vec.length > 0)

return this.vec.pop();

else

return null;

}

}

Para probar estas dos clases definimos un objeto de la clase PilaEnteros e insertamos tres valores y luego extraemos uno:

let pila1=new PilaEnteros();

pila1.insertar(20);

pila1.insertar(43);

pila1.insertar(1);

console.log(pila1.extraer()); //se imprime un 1 ya que el último en entrar es el primero en salir en una pila.

De forma similar probamos creando un objeto de la clase PilaStrings:

let pila2=new PilaStrings();

pila2.insertar('juan');

pila2.insertar('ana');

pila2.insertar('luis');

console.log(pila2.extraer()); //se imprime 'luis'

Hasta este momento no hemos presentado ninguna novedad con respecto a lo que conocemos. Veamos ahora como podemos resolver este problema pero empleando una clase genérica:

class PilaGenerica<T>{

private vec: T[] = [];

insertar(x: T) {

this.vec.push(x);

}

extraer() {

if (this.vec.length > 0)

return this.vec.pop();

else

return null;

}

}

let pila3: PilaGenerica<number>;

pila3 = new PilaGenerica<number>();

pila3.insertar(20);

pila3.insertar(42);

pila3.insertar(1);

console.log(pila3.extraer());

let pila4: PilaGenerica<string>;

pila4 = new PilaGenerica<string>();

pila4.insertar('juan');

pila4.insertar('ana');

pila4.insertar('luis');

console.log(pila4.extraer());

Como vemos hemos declarado una sola clase llamada PilaGenerica y hemos sustituido en los lugares donde hacíamos referencia a number o string por el tipo 'T' que también tenemos que hacer referencia en donde declaramos la clase:

class PilaGenerica<T>{

private vec: T[] = [];

insertar(x: T) {

this.vec.push(x);

}

extraer() {

if (this.vec.length > 0)

return this.vec.pop();

else

return null;

}

}

Luego cuando creamos un objeto de la clase PilaGenerica debemos indicar cuando la creamos el tipo de datos que administrará nuestra pila:

let pila4:PilaGenerica<string>;

pila4=new PilaGenerica<string>();

Podemos crear objetos de la clase PilaGenerica con cualquier tipo de dato primitivo (number, string, boolean etc.) o de otra clase.

Problemos de codificar otro programa que cree una pila de otra clase creada por nosotros:

class PilaGenerica<T>{

private vec: T[] = [];

insertar(x: T) {

this.vec.push(x);

}

extraer() {

if (this.vec.length > 0)

return this.vec.pop();

else

return null;

}

}

class Persona {

constructor(public nombre: string, public edad: number) { }

}

let pila5: PilaGenerica<Persona>;

pila5 = new PilaGenerica<Persona>();

pila5.insertar(new Persona('pedro', 33));

pila5.insertar(new Persona('maria', 33));

pila5.insertar(new Persona('marcos', 33));

console.log(pila5.extraer());

Creamos un objeto de la clase PilaGenerica e indicamos que almacenará objetos de la clase Persona:

let pila5:PilaGenerica<Persona>;

pila5=new PilaGenerica<Persona>();

Creamos tres objetos de la clase Persona y los insertamos en la Pila:

pila5.insertar(new Persona('pedro', 33));

pila5.insertar(new Persona('maria', 33));

pila5.insertar(new Persona('marcos', 33));

Si extraemos un elemento de la pila tenemos la última persona ingresada:

console.log(pila5.extraer());

El planteo de clases genéricas nos reduce tener que crear múltiples clases para administrar distintos tipos de datos.

58 - TypeScript y Angular: definición de una interfaz para controlar los tipos

En Angular los modelos de datos se los representa generalmente con una interfaz, veamos un ejemplo utilizando una interfaz para definir el modelo de datos.

Problema

Implementar una aplicación que permita almacenar en un arreglo los sitios favoritos de internet y una descripción del sitio.

Para resolver el problema vamos a definir la siguiente interfaz:

export interface Sitio {

url: string;

descripcion: string

}

La aplicación debe almacenar en el localStorage del navegador todos los datos para evitar que se pierdan cuando cerremos la aplicación.

Si ejecutamos la aplicación tenemos una pantalla similar a:



Podemos probar la aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto037/).

Como primer paso pasemos a crear nuestro proyecto:

* Crearemos primero el proyecto
* ng new proyecto037
* El segundo paso es crear un archivo en la carpeta 'app' con el nombre 'sitio.ts' y en su interior codificamos la interfaz 'Sitio', la idea de separarla en un archivo independiente es porque la vamos a utilizar a la interfaz 'Sitio' en más de un archivo:

sitio.ts

export interface Sitio {

url: string;

descripcion: string

}

Disponemos la palabra clave 'export' para poder importarla en otros archivos.

* Crearemos el servicio que se va a encargar de almacenar y recuperar los datos del localStorage.

Para crear el servicio desde Angular CLI ejecutamos:

ng generate service sitios

Se nos generan 2 archivos y el que debemos modificar es el archivo sitios.service.ts.

sitios.service.ts

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

import { Sitio } from './sitio';

@Injectable({

providedIn: 'root',

})

export class SitioService {

private localStorageNombre = 'sitios';

obtenerSitios(): Sitio[] {

const sitiosStr = localStorage.getItem(this.localStorageNombre);

return sitiosStr ? JSON.parse(sitiosStr) : [];

}

agregarSitio(sitio: Sitio): void {

const sitios = this.obtenerSitios();

sitios.push(sitio);

localStorage.setItem(this.localStorageNombre, JSON.stringify(sitios));

}

borrarSitio(url: string): void {

const sitios = this.obtenerSitios();

const index = sitios.findIndex((s) => s.url === url);

if (index !== -1) {

sitios.splice(index, 1);

localStorage.setItem(this.localStorageNombre, JSON.stringify(sitios));

}

}

modificarSitio(sitio: Sitio, urlBuscar:string): void {

const sitios = this.obtenerSitios();

const index = sitios.findIndex((s) => s.url === urlBuscar);

if (index !== -1) {

sitios[index] = sitio;

localStorage.setItem(this.localStorageNombre, JSON.stringify(sitios));

}

}

}

obtenerSitios(): Sitio[] {

const sitiosStr = localStorage.getItem(this.localStorageNombre);

return sitiosStr ? JSON.parse(sitiosStr) : [];

}

El método 'obtenerSitios' retorna un arreglo de la interfaz 'Sitio'. Lo primero que hacemos es con el método getItem recuperamos los datos del localStorage o nos retorna null si todavía no hay nada almacenado.

Finalmente retornamos un arreglo vacío [] si la variable 'sitioStr' almacena null o retornamos un objeto JSON a partir del string almacenado en el localStorage.

agregarSitio(sitio: Sitio): void {

const sitios = this.obtenerSitios();

sitios.push(sitio);

localStorage.setItem(this.localStorageNombre, JSON.stringify(sitios));

}

El método agregarSitio recibe como parámetro un objeto de tipo 'Sitio'. Lo primero llama al método obtenerSitios que retorna como vimos un arreglo vacío [] o un arreglo con todos los sitios almacenados actualmente en el localStorage.

Agregamos al arreglo 'sitios' el sitio que llega como parámetro y finalmente volvemos a almacenar el arreglo en el localStorage. Recordar que en el localStorage debemos convertir el arreglo 'sitios' a string mediante el método JSON.stringify.

borrarSitio(url: string): void {

const sitios = this.obtenerSitios();

const index = sitios.findIndex((s) => s.url === url);

if (index !== -1) {

sitios.splice(index, 1);

localStorage.setItem(this.localStorageNombre, JSON.stringify(sitios));

}

}

Primero recuperamos todos los sitios almacenados en el localStorage, buscamos la 'url' del sitio que queremos borrar llamando al método 'findIndex' de la clase Array. Si lo encontramos procedemos a eliminar un elemento del arreglo llamando al método 'splice' y finalmente volvemos a guardar los datos del arreglo en el localStorage.

modificarSitio(sitio: Sitio, urlBuscar:string): void {

const sitios = this.obtenerSitios();

const index = sitios.findIndex((s) => s.url === urlBuscar);

if (index !== -1) {

sitios[index] = sitio;

localStorage.setItem(this.localStorageNombre, JSON.stringify(sitios));

}

}

Buscamos la url del sitio que queremos modificar, y en caso de encontrarlo procedemos a asignar a dicha componente los datos modificados que llegan en el parámetro 'sitio'.

* Pasemos a codificar la lógica de la componente, la cual se apoya en el servicio para recuperar, agregar y modificar datos.

app.component.ts

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

import { RouterOutlet } from '@angular/router';

import { Sitio } from './sitio';

import { SitioService } from './sitios.service';

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

@Component({

selector: 'app-root',

imports: [RouterOutlet, FormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

sitio: Sitio = { url: '', descripcion: '' }; // datos cargados en el formulario

sitios: Sitio[] = []; // arreglo con todos los sitios

sitioSeleccionado: string = ''; // se almacena el sitio seleccionado con el boton "seleccionar"

constructor(private sitioService: SitioService) {

this.sitios = sitioService.obtenerSitios();

}

borrar(url: string): void {

this.sitioService.borrarSitio(url);

this.sitios = this.sitioService.obtenerSitios();

}

agregar(): void {

if (this.sitio.url.trim() === '') {

alert('Debe ingresar una URL de sitio');

return;

}

if (this.sitios.some((s) => s.url === this.sitio.url)) {

alert('Ya existe un sitio con dicha URL');

return;

}

this.sitioService.agregarSitio(this.sitio);

this.sitio = { url: '', descripcion: '' };

this.sitios = this.sitioService.obtenerSitios();

}

seleccionar(sitio: Sitio): void {

this.sitioSeleccionado = sitio.url; // Guardar la URL seleccionada

this.sitio.url = sitio.url;

this.sitio.descripcion = sitio.descripcion;

}

modificar(): void {

if (this.sitio.url.trim() === '') {

alert('Debe ingresar una URL de sitio');

return;

}

const sitioExistente = this.sitios.find((s) => s.url === this.sitio.url && s.url !== this.sitioSeleccionado);

if (sitioExistente) {

alert('Ya existe un sitio con dicha URL');

return;

}

if (this.sitioSeleccionado) {

const index = this.sitios.findIndex((s) => s.url === this.sitioSeleccionado);

if (index !== -1) {

this.sitios[index] = { ...this.sitio };

this.sitioService.modificarSitio(this.sitio,this.sitioSeleccionado);

this.sitio = { url: '', descripcion: '' };

this.sitios = this.sitioService.obtenerSitios();

this.sitioSeleccionado = '';

}

}

}

}

sitio: Sitio = { url: '', descripcion: '' }; // datos cargados en el formulario

La propiedad 'sitio' es de la interfaz 'Sitio' y lo inicializamos con string vacíos tanto para la url como la descripción. Estos dos atributos están asociados a los controles input del formulario.

sitios: Sitio[] = []; // arreglo con todos los sitios

Corresponde al arreglo con todos los sitios. En el constructor se inicializa con los sitios almacenados hasta este momento en el localStorage.

sitioSeleccionado: string = ''; // se almacena el sitio seleccionado con el boton "seleccionar"

Esta propiedad se requiere para el proceso de modificación de datos.

constructor(private sitioService: SitioService) {

this.sitios = sitioService.obtenerSitios();

}

El constructor se le inyecta el servicio y se crea en el mismo parámetro la propiedad 'sitioService'. Además llamamos al método obtenerSitios para recuperar todos los datos almacenados en el localStorage. Como sabemos el método retorna un arreglo de tipo Sitio.

borrar(url: string): void {

this.sitioService.borrarSitio(url);

this.sitios = this.sitioService.obtenerSitios();

}

El método borrar se lo llama desde la plantilla cuando se presiona el botón 'borrar' y llega como dato la 'url' del sitio a borrar. Llamamos al método borrarSitio del servicio sitioService que actualiza el localStorage eliminando dicho sitio. Además volvemos a llamar al método obtenerSitios del servicio para que nos retorne actualizado todos los sitios almacenados en el localStorage.

agregar(): void {

if (this.sitio.url.trim() === '') {

alert('Debe ingresar una URL de sitio');

return;

}

if (this.sitios.some((s) => s.url === this.sitio.url)) {

alert('Ya existe un sitio con dicha URL');

return;

}

this.sitioService.agregarSitio(this.sitio);

this.sitio = { url: '', descripcion: '' };

this.sitios = this.sitioService.obtenerSitios();

}

El método agregar hace algunas validaciones como que hayamos ingresado una 'url' en el formulario o que dicha 'url' ya si haya cargado en el arreglo.

Si pasa las validaciones llamamos al método agregarSitio del servicio sitioService y volvemos a actualizar el arreglo llamando a obtenerSitios.

seleccionar(sitio: Sitio): void {

this.sitioSeleccionado = sitio.url; // Guardar la URL seleccionada

this.sitio.url = sitio.url;

this.sitio.descripcion = sitio.descripcion;

}

El método seleccionar se llama cuando se presiona en la plantilla el botón 'seleccionar', donde procedemos a almacenar en la propiedad sitioSeleccionado la url del sitio seleccionado. También actualizamos el formulario asignando valores al atributo 'sitio'.

modificar(): void {

if (this.sitio.url.trim() === '') {

alert('Debe ingresar una URL de sitio');

return;

}

const sitioExistente = this.sitios.find((s) => s.url === this.sitio.url && s.url !== this.sitioSeleccionado);

if (sitioExistente) {

alert('Ya existe un sitio con dicha URL');

return;

}

if (this.sitioSeleccionado) {

const index = this.sitios.findIndex((s) => s.url === this.sitioSeleccionado);

if (index !== -1) {

this.sitios[index] = { ...this.sitio };

this.sitioService.modificarSitio(this.sitio,this.sitioSeleccionado);

this.sitio = { url: '', descripcion: '' };

this.sitios = this.sitioService.obtenerSitios();

this.sitioSeleccionado = '';

}

}

}

Finalmente el método modificar luego de hacer algunas validaciones procedemos a buscar el sitio que seleccionamos anteriormente y proceder a llamar al método modificarSitio del objeto sitioService pasando como parámetos los datos del formulario que se encuentran en el atributo 'sitio' y la url del sitio que anteriormente seleccionamos y almacenamos en la propiedad sitioSeleccionado.

* La plantilla muestra la tabla HTML con los sitios web y un formulario para la carga de sitios.

app.component.html

<div>

<h1>Administración de sitios web</h1>

<table>

<thead>

<tr>

<th>URL</th>

<th>Descripcion</th>

<th>Borrar</th>

<th>Seleccionar</th>

</tr>

</thead>

<tbody>

@for(sitio of sitios; track sitio.url) {

<tr>

<td><a href="https://www.{{sitio.url}}" target="\_blank">{{sitio.url}}</a></td>

<td>{{sitio.descripcion}}</td>

<td><button (click)="borrar(sitio.url)">Borrar?</button></td>

<td><button (click)="seleccionar(sitio)">Seleccionar</button></td>

</tr>

} @empty {

<tr>

<td colspan="4">

No hay sitios.

</td>

</tr>

}

</tbody>

</table>

<div>

<p>

URL:<input type="text" [(ngModel)]="sitio.url" />

</p>

<p>

Descripcion:<input type="text" [(ngModel)]="sitio.descripcion" />

</p>

<p>

<button (click)="agregar()">Agregar</button>

<button (click)="modificar()">Modificar</button>

</p>

</div>

</div>

<router-outlet />

@for(sitio of sitios; track sitio.url) {

<tr>

<td><a href="https://www.{{sitio.url}}" target="\_blank">{{sitio.url}}</a></td>

<td>{{sitio.descripcion}}</td>

<td><button (click)="borrar(sitio.url)">Borrar?</button></td>

<td><button (click)="seleccionar(sitio)">Seleccionar</button></td>

</tr>

} @empty {

<tr>

<td colspan="4">

No hay sitios.

</td>

</tr>

}

Utilizamos la estructura repetitiva '@for' en la plantilla para mostrar el nombre del sitio, su descripción y dos botones que nos permiten borrar o seleccionar el sitio.

Si el arreglo 'sitios' está vacío se ejecuta el bloque @empty.

* La hoja de estilo de la componente.

app.component.css

table {

width: 100%;

border-collapse: collapse;

margin-bottom: 20px;

}

th, td {

border: 1px solid #dddddd;

padding: 8px;

text-align: left;

}

tr:nth-child(even) {

background-color: #f2f2f2;

}

th {

background-color: #4CAF50;

color: white;

}

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto037/).

59 - TypeScript: decoradores

En TypeScript, un decorador es una función especial que se utiliza para modificar o extender el comportamiento de clases, métodos, propiedades o parámetros.

Las funciones decoradoras permiten agregar metadatos y comportamientos adicionales.

Las funciones decoradoras se ejecutan en Angular en tiempo de compilación de la aplicación, recordemos que el framework Angular compila el código TypeScript, HTML y CSS, generando JavaScript que es lo que puede interpretar un navegador web.

Veremos con dos ejemplo cual es la sintaxis para crear funciones decoradoras en TypeScript y en los próximos conceptos veremos las funciones decoradoras que vienen en el framework de Angular.

Función decoradora de clase sin parámetros.

Podemos agregar el código siguiente a un proyecto que tengamos en Angular:

app.component.ts

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

import { RouterOutlet } from '@angular/router';

function MostrarMensajeDeCreacion(constructor: Function) {

console.log(constructor.toString());

const prototipo = constructor.prototype;

const nombresMetodos = Object.getOwnPropertyNames(prototipo)

.filter(nombre => typeof prototipo[nombre] === 'function');

console.log('Métodos:', nombresMetodos.join(', '));

}

// Aplicamos el decorador a una clase

@MostrarMensajeDeCreacion

class MiClase {

constructor() {

console.log('Objeto creado');

}

metodo1() {

console.log('Metodo 1 ejecutado');

}

metodo2() {

console.log('Metodo 2 ejecutado');

}

}

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent{

title = 'prueba';

}

Para aplicar una función decoradora a una clase debemos anteceder al nombre de la clase el caracter @ seguido del nombre de la función decoradora:

@MostrarMensajeDeCreacion

class MiClase {

La función decoradora MostrarMensajeDeCreacion llega una función (el constructor de la clase) como parámetro. En este caso, la función imprime la representación en cadena de la clase y los nombres de los métodos de la clase:

function MostrarMensajeDeCreacion(constructor: Function) {

console.log(constructor.toString());

const prototipo = constructor.prototype;

const nombresMetodos = Object.getOwnPropertyNames(prototipo)

.filter(nombre => typeof prototipo[nombre] === 'function');

console.log('Métodos:', nombresMetodos.join(', '));

}

No hemos definido objetos de la clase 'MiClase' y podemos ver que el compilador de Angular ejecuta el decorador cuando compila la aplicación:



Es solo un ejemplo elemental para entender que la función decoradora se ejecuta independientemente a que definamos objetos de la clase. El decorador se ejecuta en tiempo de compilación y se utiliza para modificar o extender la funcionalidad de la clase, nosotros solo hemos hecho unas pocas salidas por la consola.

Función decoradora de clase con parámetros.

Podemos pasar parámetros a una función decoradora, veamos un ejemplo donde le pasamos un objeto de una determinada interface (en forma similar el framework Angular utiliza los decoradores para añadir funcionalidades por ejemplo a las componentes):

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

import { RouterOutlet } from '@angular/router';

// Interfaz para el parámetro del decorador

interface DetallesDeCreacion {

selector: string;

templateUrl: string;

styleUrl: string;

}

// Decorador de Clase para Mostrar Detalles de Creación con Parámetro de la Interfaz

function MostrarDetallesDeCreacion(detalles: DetallesDeCreacion) {

return function (constructor: Function) {

console.log(constructor.toString());

// Mostrar información adicional sobre métodos y propiedades

const prototipo = constructor.prototype;

const nombresMetodos = Object.getOwnPropertyNames(prototipo)

.filter(nombre => typeof prototipo[nombre] === 'function');

console.log('Métodos:', nombresMetodos.join(', '));

console.log('Template URL:', detalles.templateUrl);

console.log('Style URL:', detalles.styleUrl);

};

}

// Aplicamos el decorador a una clase con un parámetro de la interfaz

@MostrarDetallesDeCreacion({

selector: 'app-mi-clase',

templateUrl: 'mi-clase.component.html',

styleUrl: 'mi-clase.component.css',

})

class MiClase {

constructor() {

console.log('Objeto creado');

}

metodo1() {

console.log('Metodo 1 ejecutado');

}

metodo2() {

console.log('Metodo 2 ejecutado');

}

}

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'prueba';

}

interface DetallesDeCreacion {

selector: string;

templateUrl: string;

styleUrl: string;

}

Se define una interfaz llamada DetallesDeCreacion que especifica tres propiedades: selector, templateUrl, y styleUrl. Es solo un ejemplo y los nombres de las propiedades pueden ser cualquiera.

function MostrarDetallesDeCreacion(detalles: DetallesDeCreacion) {

return function (constructor: Function) {

console.log(constructor.toString());

// Mostrar información adicional sobre métodos y propiedades

const prototipo = constructor.prototype;

const nombresMetodos = Object.getOwnPropertyNames(prototipo)

.filter(nombre => typeof prototipo[nombre] === 'function');

console.log('Métodos:', nombresMetodos.join(', '));

console.log('Template URL:', detalles.templateUrl);

console.log('Style URL:', detalles.styleUrl);

};

}

Se define una función llamada MostrarDetallesDeCreacion que toma un parámetro 'detalles' de tipo DetallesDeCreacion. Esta función devuelve otra función que actúa como el decorador real.

Dentro de la función del decorador, se imprime la representación en cadena del constructor de la clase, así como los nombres de los métodos de la clase.

Luego, imprime las propiedades del parámetro 'detalles' que fueron proporcionadas al aplicar el decorador.

// Aplicamos el decorador a una clase con un parámetro de la interfaz

@MostrarDetallesDeCreacion({

selector: 'app-mi-clase',

templateUrl: 'mi-clase.component.html',

styleUrl: 'mi-clase.component.css',

})

class MiClase {

constructor() {

console.log('Objeto creado');

}

metodo1() {

console.log('Metodo 1 ejecutado');

}

metodo2() {

console.log('Metodo 2 ejecutado');

}

}

Se aplica el decorador MostrarDetallesDeCreacion a la clase MiClase con un objeto que cumple con la interfaz DetallesDeCreacion. Los valores proporcionados (como selector, standalone, templateUrl, y styleUrl) se utilizarán dentro del decorador para mostrar información adicional.

Tengamos en cuenta que no creamos objetos de la clase 'MiClase' y el compilador de Angular ejecuta la función decoradora aplicada a la clase:



Este ejemplo nos debe dar una idea como el compilador de Angular procesa el código de la componente AppComponent:

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'prueba';

}

Lo primero que hace es ejecutar la función decoradora @Component y le pasa como parámetro un objeto, en este caso inicializando 4 propiedades: selector, imports, templateUrl y styleUrls. Todos estos datos complementan a la clase AppComponent para acceder a los archivos: app.component.html y app.component.css

60 - TypeScript y Angular: decorador @Component

En Angular, el decorador @Component es esencial para la creación y configuración de componentes. Este decorador se utiliza para asociar metadatos con la clase de un componente, proporciona información sobre su comportamiento, aspecto y configuración.

Los metadatos incluyen propiedades como selector (identificador HTML para el componente), template (código HTML o referencia a un archivo HTML), styleUrls (archivos de estilo o estilos en línea) etc.

Crearemos un proyecto para probar distintas variantes sobre la configuración de los parámetros del decorador @Component.

ng new proyecto038

Como ya sabemos Angular CLI al crear un proyecto nos crea una primer componente llamada AppComponent:

app.component.ts

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'proyecto038';

}

Ahora que ya conocemos mejor el concepto de decorador, pasemos a ver las distintas propiedades del objeto que le pasamos a la función Component.

La propiedad 'selector' hace referencia a la etiqueta que debemos agregar a nuestra página para crear una componente de dicho tipo:

selector: 'app-root',

En este caso se encuentra en el archivo index.html:

<body>

<app-root></app-root>

</body>

Otra propiedad que debemos configurar es imports, que se utiliza para declarar otras componentes que hacemos uso, o módulos que contienen componentes, directivas u otros artefactos que el componente actual necesita para funcionar correctamente.

La propiedad templateUrl en el decorador @Component se utiliza para especificar la ubicación del archivo HTML externo que contiene la plantilla del componente. En otras palabras, templateUrl se utiliza para separar el código HTML de la lógica del componente, lo que facilita la organización y mantenimiento del código.

templateUrl: './app.component.html',

Otra variante que vamos a probar es eliminar la propiedad 'templateUrl' y proceder a insertar la propiedad 'template' con la plantilla HTML directamente dentro del decorador (podemos eliminar el archivo 'app.component.html'), puede ser útil si nuestra componente es muy sencilla y queremos concentrar la plantilla HTML y la clase en el mismo archivo:

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

template: `<h1>Componente de Angular</h1>

<p>La plantilla de la componente se encuentra definida

directamente en el decorador.</p>`,

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'proyecto038';

}

De forma similar podemos definir la hoja de estilo de la componente en un archivo o un conjunto de archivos CSS separados:

styleUrls: ['./app.component.css']

O borrar la propiedad 'styleUrls' por la propiedad 'style':

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

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

imports: [RouterOutlet],

template: `<h1>Componente de Angular</h1>

<p>La plantilla de la componente se encuentra definida

directamente en el decorador.</p>`,

styles: `h1 {

color:red

}

p {

color:blue

}`

})

export class AppComponent {

title = 'proyecto038';

}

Es importante hacer notar que la función decoradora 'Component' debemos importarla para su uso (cuando creamos una componente, dicho código ya se ha agregado):

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

61 - TypeScript y Angular: decoradores @Input y @Output

Ya vimos en los primeros conceptos que el decorador @Input nos permite pasar datos de la componente padre a la componente hija. También vimos que el decorador @Output nos permite disparar eventos desde la componente hija a la componente padre.

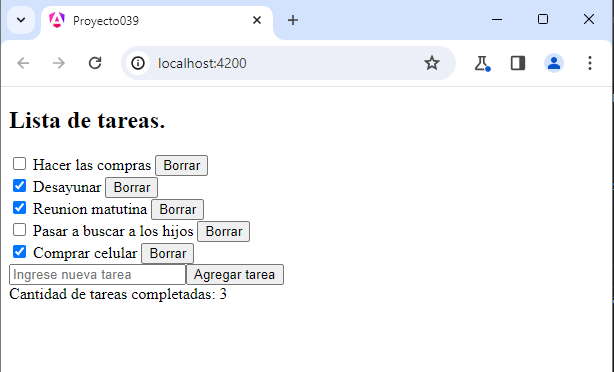
Ahora ya tenemos un conocimiento más profundo de los decoradores y como Angular en tiempo de compilación puede generar un código JavaScript y enlazar el pasaje de datos entre componentes.

Problema

Crear una aplicación que permita registrar tareas que se deben desarrollar y mediante controles checkbox tildar las ya realizadas, informando en la parte inferior cuantas de dichas tareas han sido finalizadas hasta dicho momento.

Crear dos componentes, una llamada ListaTareas que muestre todas las tareas, permita ingresar nuevas tareas e informe las tareas finalizadas y por otro lado una componente llamada Tarea que tenga por objetivo mostrar una tarea, poder indicar que ya ha sido realizada e informar a la componente padre cada vez que cambia el estado de la tarea.

La interfaz visual es similar a:



Podemos probar esta aplicación también en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto039/).

* Crearemos primero el proyecto
* ng new proyecto039
* Creamos el modelo de dato que representa una tarea mediante una 'interface':
* ng generate interface tarea

Se nos crea el archivo 'tarea.ts' donde definimos la interface:

export interface Tarea {

id: number,

descripcion: string,

finalizada: boolean

}

Por cada tarea guardaremos su identificador, descripción y si la tarea fue finalizada.

* Pasemos a crear ahora la componente "Tarea":
* ng generate component tarea

Se nos crea una carpeta con los 4 archivos que representan la componente, procedemos a modificar la clase propiamente dicha y la plantilla HTML.

tarea.component.ts

import { Component, Input, Output, EventEmitter } from '@angular/core';

import { Tarea } from '../tarea';

@Component({

selector: 'app-tarea',

imports: [],

templateUrl: './tarea.component.html',

styleUrl: './tarea.component.css'

})

export class TareaComponent {

@Input() tarea: Tarea = { id: 0, descripcion: '', finalizada: false };

@Output() borrarTarea: EventEmitter<number> = new EventEmitter<number>();

cambiarEstado() {

this.tarea.finalizada = !this.tarea.finalizada;

}

borrar() {

this.borrarTarea.emit(this.tarea.id);

}

}

import { Component, Input, Output, EventEmitter } from '@angular/core';

Debemos importar primero los 3 decoradores y la interface EventEmitter.

import { Tarea } from '../tarea';

También importamos la interface que creamos nosotros.

@Input() tarea: Tarea = { id: 0, descripcion: '', finalizada: false };

Definimos el decorador input que se aplica a la propiedad 'tarea' (es decir es un decorador de propiedad y no de clase como Component), el mismo espera recibir una tarea como entrada, lo inicializamos pero seguramente cuando lo invoquemos cambie con los datos que llegan desde la componente padre.

@Output() borrarTarea: EventEmitter<number> = new EventEmitter<number>();

Es un decorador que indica que borrarTarea es un evento de salida. Cuando se llama al método borrar(), emite un evento con el ID de la tarea que se va a borrar.

cambiarEstado() {

this.tarea.finalizada = !this.tarea.finalizada;

}

Es un método que cambia el estado de la tarea (marcada o no marcada como finalizada) al ser llamado.

borrar() {

this.borrarTarea.emit(this.tarea.id);

}

Emite el evento borrarTarea con el ID de la tarea que se va a borrar.

La plantilla HTML para 'TareaComponent' es:

<div>

<input type="checkbox" [checked]="tarea.finalizada" (change)="cambiarEstado()">

{{ tarea.descripcion }}

<button (click)="borrar()">Borrar</button>

</div>

<input type="checkbox" [checked]="tarea.finalizada" (change)="cambiarEstado()">

Se crea un elemento input de tipo checkbox. [checked]="tarea.finalizada" utiliza la propiedad binding para enlazar el estado del checkbox al valor de tarea 'finalizada'. Esto significa que el checkbox estará marcado (checked) si tarea.finalizada es true, y desmarcado si es false.

(change)="cambiarEstado()": Utiliza un evento de cambio (change) para llamar al método cambiarEstado() cuando el estado del checkbox cambia.

<button (click)="borrar()">Borrar</button>

(click)="borrar()": Utiliza un evento de clic (click) para llamar al método borrar()

* Pasemos a crear ahora la componente "listatareas":
* ng generate component listatareas

Se nos crea una carpeta con los 4 archivos que representan la componente, procedemos a modificar la clase propiamente dicha y la plantilla HTML.

listatareas.component.ts

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

import { Tarea } from '../tarea';

import { TareaComponent } from '../tarea/tarea.component';

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

@Component({

selector: 'app-listatareas',

imports: [TareaComponent, FormsModule],

templateUrl: './listatareas.component.html',

styleUrl: './listatareas.component.css'

})

export class ListatareasComponent {

tareas: Tarea[] = [];

descripcionTarea: string = '';

agregarTarea() {

const nuevaTarea: Tarea = { id: this.tareas.length + 1, descripcion: this.descripcionTarea, finalizada: false };

this.tareas.push(nuevaTarea);

this.descripcionTarea = '';

}

borrarTarea(tareaId: number) {

this.tareas = this.tareas.filter(tarea => tarea.id !== tareaId);

}

cantidadTareasCompletadas(): number {

return this.tareas.filter(tarea => tarea.finalizada).length;

}

}

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

import { Tarea } from '../tarea';

import { TareaComponent } from '../tarea/tarea.component';

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

Importamos el decorador Component, la interface Tarea, la componente TareaComponent y como vamos a trabajar con formularios basado en plantillas debemos importar FormsModule.

@Component({

selector: 'app-listatareas',

imports: [TareaComponent, FormsModule],

templateUrl: './listatareas.component.html',

styleUrl: './listatareas.component.css'

})

Importamos en el decorador de la componente la componente que creamos nosotros 'TareaComponent' y FormsModule.

tareas: Tarea[] = [];

tareas es un array que almacenará las instancias de la interface Tarea.

descripcionTarea: string = '';

descripcionTarea es una propiedad que se utiliza para almacenar la descripción de la tarea que se va a agregar y esta vinculado al control input de tipo type de la plantilla HTML.

agregarTarea() {

const nuevaTarea: Tarea = { id: this.tareas.length + 1, descripcion: this.descripcionTarea, finalizada: false };

this.tareas.push(nuevaTarea);

this.descripcionTarea = '';

}

Es un método que crea una nueva tarea con la descripción ingresada y la agrega al array de tareas. Luego, reinicia la propiedad descripcionTarea para que se borre en el formulario la tarea ingresada.

borrarTarea(tareaId: number) {

this.tareas = this.tareas.filter(tarea => tarea.id !== tareaId);

}

Es un método que filtra las tareas para eliminar aquella cuyo id coincide con el tareaId proporcionado desde la plantilla HTML.

cantidadTareasCompletadas(): number {

return this.tareas.filter(tarea => tarea.finalizada).length;

}

Devuelve el número de tareas completadas, utilizando el método filter para filtrar las tareas con finalizada establecida como true y luego calcula la longitud del array resultante.

La plantilla HTML para 'ListatareasComponent' es:

<div>

<h2>Lista de tareas.</h2>

@for(tarea of tareas; track tarea.id) {

<app-tarea [tarea]="tarea" (borrarTarea)="borrarTarea($event)"></app-tarea>

}

<div>

<input [(ngModel)]="descripcionTarea" placeholder="Ingrese nueva tarea">

<button (click)="agregarTarea()">Agregar tarea</button>

</div>

</div>

<div>

Cantidad de tareas completadas: {{ cantidadTareasCompletadas() }}

</div>

@for(tarea of tareas; track tarea.id) {

Utilizamos un bucle para recorrer el arreglo de tareas.

<app-tarea [tarea]="tarea" (borrarTarea)="borrarTarea($event)"></app-tarea>

Para cada tarea, se utiliza el componente app-tarea para representarla. Se enlaza la tarea actual mediante [tarea]="tarea", y se escucha el evento borrarTarea para ejecutar el método.

<input [(ngModel)]="descripcionTarea" placeholder="Ingrese nueva tarea">

Crea un campo de entrada de texto con enlace bidireccional ([(ngModel)]) a la propiedad descripcionTarea. Esto permite que la variable descripcionTarea en el componente refleje los cambios en este campo de entrada.

<button (click)="agregarTarea()">Agregar tarea</button>

Crea un botón que llama al método agregarTarea() cuando se hace clic en él. Este método agrega una nueva tarea a la lista.

* Por último debemos modificar la componente que crea Angular por defecto, donde agregaremos la lista de tareas:

appcomponent.component.ts

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

import { RouterOutlet } from '@angular/router';

import { ListatareasComponent } from './listatareas/listatareas.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ListatareasComponent],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

}

import { ListatareasComponent } from './listatareas/listatareas.component';

Importa la componente ListatareasComponent.

@Component({

selector: 'app-root',

imports: [RouterOutlet, ListatareasComponent],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

También en el decorador hacemos referencia a la importación del componente 'ListatareasComponent'.

La plantilla HTML para 'AppComponent' es:

<app-listatareas></app-listatareas>

<router-outlet />

Incluimos la componente ListatareasComponent en la plantilla HTML mediante la etiqueta app-listatareas que se encuentra definida en la propidad select del decorador de la clase 'ListatareasComponent'.

* Ahora podemos ejecutar la aplicación:
* ng serve -o

Podemos probar esta aplicación también en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto039/).

62 - TypeScript y Angular: decorador @Injectable

El decorador @Injectable se utiliza para marcar una clase como un servicio inyectable. Cuando un servicio es inyectable, significa que puede ser incluido en la lista de dependencias y Angular puede proporcionar instancias de ese servicio a otras clases mediante la inyección de dependencias.

Angular utiliza un sistema de inyección de dependencias para gestionar las dependencias entre diferentes partes de una aplicación. Esto permite la creación de instancias de clases y su provisión a otras clases que las necesitan.

Se aplica el decorador @Injectable a la clase del servicio que se desea inyectar. Este decorador no requiere ninguna configuración adicional y se coloca en la parte superior de la declaración de la clase.

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

@Injectable({

providedIn: 'root',

})

export class MiServicio {

// Métodos y propiedades del servicio

}

@Injectable está configurado con { providedIn: 'root' } Esto significa que Angular proporcionará una única instancia de MiServicio en toda la aplicación (En ningún momento nosotros debemos instanciar la clase MiServicio)

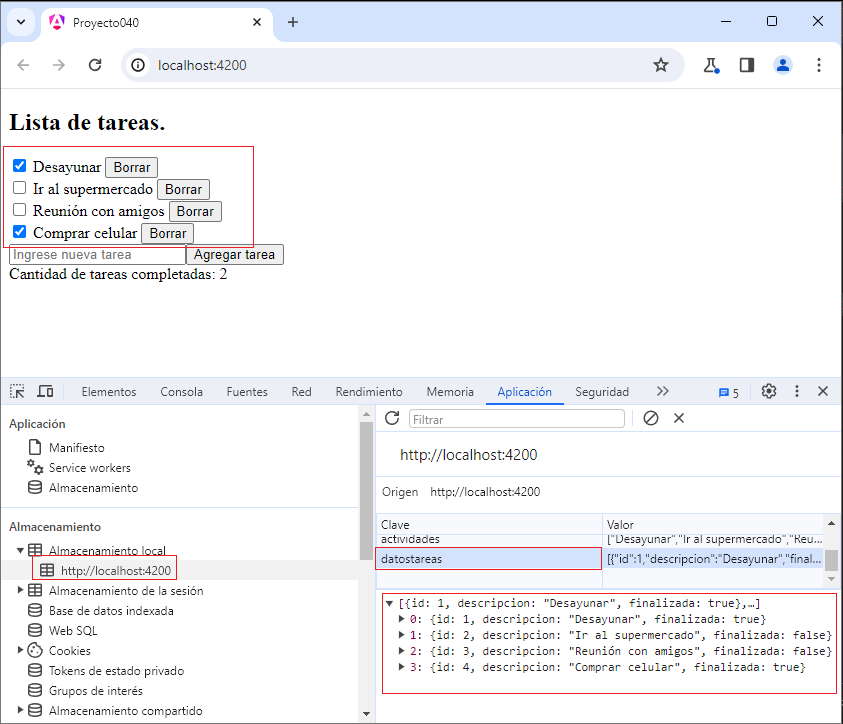
Problema

Vamos a resolver el mismo problema que el concepto anterior, pero vamos a almacenar las tareas en el localStorage.

Recordemos el problema: crear una aplicación que permita registrar tareas que se deben desarrollar y mediante controles checkbox tildar las ya realizadas, informando en la parte inferior cuantas de dichas tareas han sido finalizadas hasta dicho momento.

Crear dos componentes, una llamada ListaTareas que muestre todas las tareas, permita ingresar nuevas tareas e informe las tareas finalizadas y por otro lado una componente llamada Tarea que tenga por objetivo mostrar una tarea, poder indicar que ya ha sido realizada e informar a la componente padre cada vez que cambia el estado de la tarea.

La interfaz visual es similar a:



Podemos probar esta aplicación también en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto040/).

* Crearemos primero el proyecto
* ng new proyecto040
* Creamos el modelo de dato que representa una tarea mediante una 'interface':
* ng generate interface tarea

Se nos crea el archivo 'tarea.ts' donde definimos la interface:

export interface Tarea {

id: number,

descripcion: string,

finalizada: boolean

}

Por cada tarea guardaremos su identificador, descripción y si la tarea fue finalizada.

* Crearemos ahora el servicio que nos va a permitir almacenar, recuperar y modificar las tareas en el localStorage del navegador:
* ng generate service tareas

Procedemos a crear la lógica del servicio:

tareas.service.ts

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

import { Tarea } from './tarea';

@Injectable({

providedIn: 'root',

})

export class TareaService {

private localStorageKey = 'datostareas';

obtenerTareas(): Tarea[] {

const tareasStr = localStorage.getItem(this.localStorageKey);

return tareasStr ? JSON.parse(tareasStr) : [];

}

guardarTareas(tareas: Tarea[]): void {

localStorage.setItem(this.localStorageKey, JSON.stringify(tareas));

}

agregarTarea(nuevaTarea: Tarea): void {

const tareas = this.obtenerTareas();

tareas.push(nuevaTarea);

this.guardarTareas(tareas);

}

borrarTarea(tareaId: number): void {

const tareas = this.obtenerTareas().filter(tarea => tarea.id !== tareaId);

this.guardarTareas(tareas);

}

}

* Pasemos a crear ahora la componente "Tarea":
* ng generate component tarea

Se nos crea una carpeta con los 4 archivos que representan la componente, procedemos a modificar la clase propiamente dicha y la plantilla HTML.

tarea.component.ts

import { Component, Input, Output, EventEmitter } from '@angular/core';

import { Tarea } from '../tarea';

@Component({

selector: 'app-tarea',

imports: [],

templateUrl: './tarea.component.html',

styleUrl: './tarea.component.css'

})

export class TareaComponent {

@Input() tarea: Tarea = { id: 0, descripcion: '', finalizada: false };

@Output() borrarTarea: EventEmitter<number> = new EventEmitter<number>();

@Output() actualizarFinalizada: EventEmitter<Tarea> = new EventEmitter<Tarea>();

cambiarEstado() {

this.tarea.finalizada = !this.tarea.finalizada;

console.log(" mi")

this.actualizarFinalizada.emit(this.tarea);

}

borrar() {

this.borrarTarea.emit(this.tarea.id);

}

}

La plantilla HTML para 'TareaComponent' es:

<div>

<input type="checkbox" [checked]="tarea.finalizada" (change)="cambiarEstado()">

{{ tarea.descripcion }}

<button (click)="borrar()">Borrar</button>

</div>

* Pasemos a crear ahora la componente "listatareas":
* ng generate component listatareas

Se nos crea una carpeta con los 4 archivos que representan la componente, procedemos a modificar la clase propiamente dicha y la plantilla HTML.

listatareas.component.ts

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

import { Tarea } from '../tarea';

import { TareaComponent } from '../tarea/tarea.component';

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

import { TareaService } from '../tareas.service';

@Component({

selector: 'app-listatareas',

imports: [TareaComponent, FormsModule],

templateUrl: './listatareas.component.html',

styleUrl: './listatareas.component.css'

})

export class ListatareasComponent {

tareas: Tarea[] = [];

descripcionTarea: string = '';

constructor(private tareaService: TareaService) {

this.tareas = this.tareaService.obtenerTareas();

}

agregarTarea() {

let nuevaTarea: Tarea;

if (this.tareas.length == 0) {

nuevaTarea = { id: 1, descripcion: this.descripcionTarea, finalizada: false };

} else {

nuevaTarea = { id: this.tareas[this.tareas.length - 1].id + 1, descripcion: this.descripcionTarea, finalizada: false };

}

this.tareas.push(nuevaTarea);

this.tareaService.agregarTarea(nuevaTarea);

this.descripcionTarea = '';

}

borrarTarea(tareaId: number) {

this.tareas = this.tareas.filter(tarea => tarea.id !== tareaId);

this.tareaService.borrarTarea(tareaId);

}

actualizarFinalizada(tarea: Tarea) {

this.tareas = this.tareas.map(ta => ta.id === tarea.id ? tarea : ta)

this.tareaService.guardarTareas(this.tareas);

}

cantidadTareasCompletadas(): number {

return this.tareas.filter(tarea => tarea.finalizada).length;

}

}

La plantilla HTML para 'ListatareasComponent' es:

<div>

<h2>Lista de tareas.</h2>

@for(tarea of tareas; track tarea.id) {

<app-tarea [tarea]="tarea" (borrarTarea)="borrarTarea($event)"

(actualizarFinalizada)="actualizarFinalizada(tarea)"></app-tarea>

}

<div>

<input [(ngModel)]="descripcionTarea" placeholder="Ingrese nueva tarea">

<button (click)="agregarTarea()">Agregar tarea</button>

</div>

</div>

<div>

Cantidad de tareas completadas: {{ cantidadTareasCompletadas() }}

</div>

* Por último debemos modificar la componente que crea Angular por defecto, donde agregaremos la lista de tareas:

appcomponent.component.ts

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

import { RouterOutlet } from '@angular/router';

import { ListatareasComponent } from './listatareas/listatareas.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, ListatareasComponent],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

}

La plantilla HTML para 'AppComponent' es:

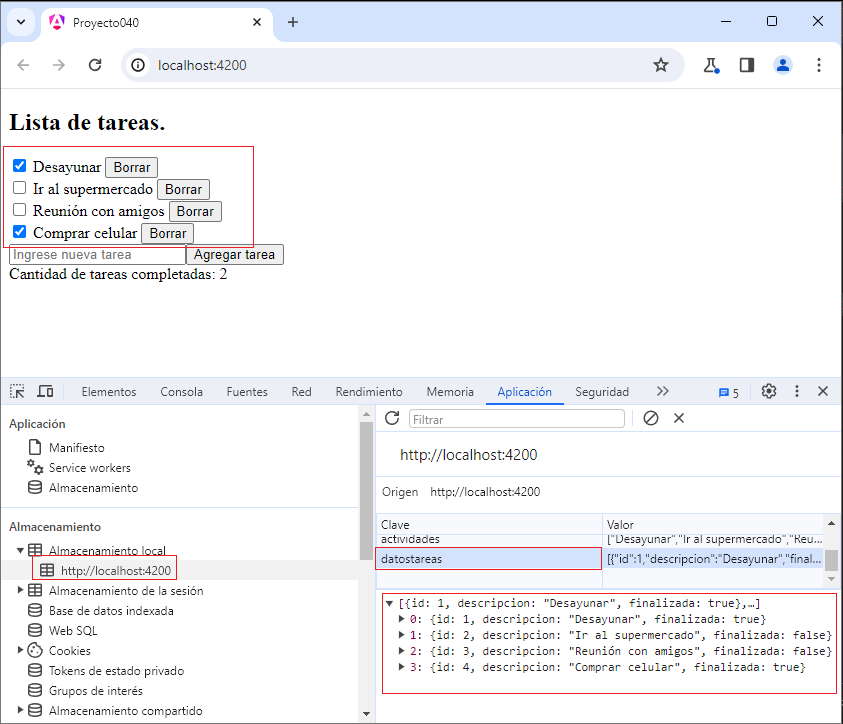
<app-listatareas></app-listatareas>

<router-outlet />

Incluimos la componente ListatareasComponent en la plantilla HTML mediante la etiqueta app-listatareas que se encuentra definida en la propidad select del decorador de la clase 'ListatareasComponent'.

* Ahora podemos ejecutar la aplicación:
* ng serve -o

Podemos ver el localStorage con los datos de las tareas registradas hasta ese momento:



Podemos probar esta aplicación también en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto040/).

63 - TypeScript y Angular: decorador @Directive

En Angular hemos visto que el decorador @Directive se utiliza para definir una directiva personalizada. Las directivas son instrucciones que se aplican al DOM y pueden modificar su estructura, apariencia o comportamiento. A través del decorador @Directive, puedes crear tu propia directiva y luego usarla en los elementos del DOM dentro de tu aplicación Angular.

Problema

Crear una directiva que cambie dinámicamente el fondo de un elemento en función del tiempo actual del día. Llamaremos a esta directiva cambiarFondoSegunHora. La idea es que el fondo del elemento cambie de color dependiendo de si es de día, tarde o noche.

* Crearemos primero el proyecto
* ng new proyecto041
* Creamos la directiva cambiarFondoSegunHora:
* ng generate directive cambiarFondoSegunHora

Se nos crea el archivo 'cambiar-fondo-segun-hora.directive.ts' donde codificamos:

import { Directive, ElementRef, Renderer2, OnInit } from '@angular/core';

@Directive({

selector: '[appCambiarFondoSegunHora]'

})

export class CambiarFondoSegunHoraDirective implements OnInit{

constructor(private el: ElementRef, private renderer: Renderer2) { }

ngOnInit() {

this.cambiarFondoSegunHora();

}

private cambiarFondoSegunHora() {

const horaActual = new Date().getHours();

let colorFondo = '';

if (horaActual >= 6 && horaActual < 12) {

colorFondo = 'yellow'; // Mañana

} else if (horaActual >= 12 && horaActual < 18) {

colorFondo = 'lightgray'; // Tarde

} else {

colorFondo = 'gray'; // Noche

}

this.renderer.setStyle(this.el.nativeElement, 'background-color', colorFondo);

}

}

@Directive({

selector: '[appCambiarFondoSegunHora]'

})

@Directive es un decorador que se utiliza para definir una directiva personalizada en Angular.

selector: '[appCambiarFondoSegunHora]',

Indica que esta directiva se aplicará a elementos del DOM con el atributo appCambiarFondoSegunHora.

constructor(private el: ElementRef, private renderer: Renderer2) { }

En el constructor, se inyectan las instancias de ElementRef y Renderer2. ElementRef proporciona acceso al elemento del DOM al que se aplica la directiva, y Renderer2 permite manipular el DOM de manera segura.

ngOnInit() {

this.cambiarFondoSegunHora();

}

El método ngOnInit() se llama automáticamente después de que Angular haya inicializado la directiva. En este caso, se utiliza para llamar al método cambiarFondoSegunHora().

private cambiarFondoSegunHora() {

const horaActual = new Date().getHours();

let colorFondo = '';

if (horaActual >= 6 && horaActual < 12) {

colorFondo = 'yellow'; // Mañana

} else if (horaActual >= 12 && horaActual < 18) {

colorFondo = 'lightgray'; // Tarde

} else {

colorFondo = 'gray'; // Noche

}

this.renderer.setStyle(this.el.nativeElement, 'background-color', colorFondo);

}

El método privado cambiarFondoSegunHora() determina la hora actual y asigna un color de fondo según la hora del día. Luego, utiliza Renderer2 para establecer el estilo del fondo del elemento.

* Ahora en la clase principal de la aplicación hacemos uso de esta directiva.

app.component.ts

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

import { RouterOutlet } from '@angular/router';

import { CambiarFondoSegunHoraDirective } from './cambiar-fondo-segun-hora.directive';

@Component({

selector: 'app-root',

imports: [RouterOutlet, CambiarFondoSegunHoraDirective],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

}

Y en la plantilla disponemos la directiva que hemos creado:

app.component.html

<div appCambiarFondoSegunHora>

Este fondo cambia según la hora del día.

</div>

<router-outlet />

Podemos probar esta aplicación también en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto041/).

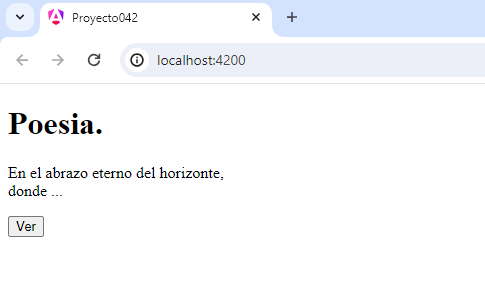
64 - TypeScript y Angular: decorador @Pipe

Vimos en conceptos anteriores que una pipe tiene por objetivo convertir un dato en la vista de la componente (html) con la finalidad que el usuario tenga una mejor experiencia, también vimos que el framework Angular provee un conjunto de pipes por defecto como pueden ser: uppercase, lowercase, json, date etc.

El decorador @Pipe aparece cuando tenemos que crear nuestras propias pipes que se adapten a resolver problemas de nuestra aplicación.

Problema

Crear una pipe que trunque un texto y agregue puntos suspensivos que da a entender que el texto ha sido truncado.



* Crearemos primero el proyecto
* ng new proyecto042
* Creamos la tubería (pipe) :
* ng generate pipe truncarTexto

Se crean 2 archivos y pasamos a codificar nuestra pipe:

truncar-texto.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'truncarTexto'

})

export class TruncarTextoPipe implements PipeTransform {

transform(texto: string, longitudMaxima: number = 50): string {

if (texto.length <= longitudMaxima) {

return texto;

}

// Truncar el texto y agregar puntos suspensivos

const textoTruncado = texto.slice(0, longitudMaxima) + '...';

return textoTruncado;

}

}

La tubería TruncarTextoPipe se utiliza para truncar un texto si supera una longitud máxima especificada.

import { Pipe, PipeTransform } from '@angular/core';

Importamos las interfaces Pipe y PipeTransform del módulo '@angular/core'.

@Pipe({

name: 'truncarTexto'

})

Como vemos utilizamos la función decoradora @Pipe y le pasamos como parámetro el nombre de nuestra tubería que haremos uso luego en una plantilla de una componente.

export class TruncarTextoPipe implements PipeTransform {

transform(texto: string, longitudMaxima: number = 50): string {

//lógica

}

}

La clase TruncarTextoPipe implementa la interfaz PipeTransform, que requiere la implementación del método transform. Este método se ejecutará cuando la tubería sea utilizada en una plantilla y toma dos parámetros:  
texto: El valor que se va a transformar (en este caso, un texto).  
longitudMaxima: La longitud máxima permitida para el texto antes de truncarlo. Por defecto, se establece en 50 caracteres si no se proporciona otro valor.

if (texto.length <= longitudMaxima) {

return texto;

}

// Truncar el texto y agregar puntos suspensivos

const textoTruncado = texto.slice(0, longitudMaxima) + '...';

return textoTruncado;

La lógica del método verifica si la longitud del texto es menor o igual a la longitud máxima especificada. Si es así, el texto no se modifica y se devuelve tal cual. Si supera la longitud máxima, se trunca utilizando slice y se le añaden puntos suspensivos ('...') al final. El texto truncado es luego devuelto.

* Este fragmento de código es un componente Angular que utiliza una tubería personalizada (TruncarTextoPipe) y manipula la visualización de un poema en función de la interacción del usuario según la presión de un botón.

app.component.ts

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

import { RouterOutlet } from '@angular/router';

import { TruncarTextoPipe } from './truncar-texto.pipe';

@Component({

selector: 'app-root',

imports: [RouterOutlet, TruncarTextoPipe],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

poesia = `<p>En el abrazo eterno del horizonte,<br>

donde el cielo y el mar se entrelazan,<br>

navegan sueños en olas danzantes,<br>

poesía salada que el viento abraza.</p>

<p>Las gaviotas pintan líneas al viento,<br>

testigos de historias que el mar relata,<br>

susurran sus secretos en cada momento,<br>

mientras las olas cantan su serenata.</p>`

completo = false;

mensajeBoton = "Ver"

cambiar() {

this.completo = !this.completo;

this.mensajeBoton = this.completo ? "Ocultar" : "Ver"

}

}

import { TruncarTextoPipe } from './truncar-texto.pipe';

Se importa la clase TruncarTextoPipe.

imports: [RouterOutlet, TruncarTextoPipe],

También debemos hacer referencia a la clase en la propiedad imports del decorador de la clase.

completo = false;

completo: Un booleano que indica si se debe mostrar el poema completo o truncado.

mensajeBoton = "Ver"

mensajeBoton: El mensaje que aparecerá en el botón para alternar entre la vista completa y la truncada.

cambiar() {

this.completo = !this.completo;

this.mensajeBoton = this.completo ? "Ocultar" : "Ver"

}

Un método que se ejecuta cuando se hace clic en el botón. Cambia el valor de completo y actualiza el mensaje del botón en consecuencia.

Y en la plantilla disponemos la llamada a la pipe que hemos creado:

app.component.html

<h1>Poesia.</h1>

@if (completo) {

<div [innerHTML]="poesia"></div>

} @else {

<div [innerHTML]="poesia|truncarTexto"></div>

}

<button (click)="cambiar()">{{mensajeBoton}}</button>

<router-outlet />

@if (completo) {

<div [innerHTML]="poesia"></div>

}

<router-outlet />

Si la propiedad 'completo' almacena un true, luego se muestra la poesía en forma completa.

[innerHTML]: Se utiliza para interpolar HTML dinámicamente, permitiendo que las etiquetas HTML en el contenido de poesia se interpreten correctamente.

@else {

<div [innerHTML]="poesia|truncarTexto"></div>

}

Por el else mostramos solo los primeros 50 caracteres contenidos en la propiedad 'poesia'.

Si queremos que se muestren solo los primeros 15 caracteres pasamos un parámetro a la pipe 'truncarTexto':

<div [innerHTML]="poesia|truncarTexto:15"></div>

Podemos probar esta aplicación también en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto042/).

65 - Angular e instalación de Bootstrap original

Bootstrap es un framework de CSS que nos facilita y estandariza el desarrollo de sitios web.

Veremos que hay varias maneras de utilizar el framework original de Bootstrap y en conceptos futuros veremos como instalar versiones de Bootstrap adaptadas directamente al ambiente de Angular.

Si todavía no ha trabajado con Bootstrap puede visitar el curso de [Bootstrap 5 Ya](https://www.tutorialesprogramacionya.com/bootstrap5ya).

Distintas formas de instalar Bootstrap 5 original.

Crearemos un proyecto nuevo para probar las distintas maneras de acceder a Bootstrap 5 desde Angular:

ng new proyecto043

1. La forma más sencilla es utilizar un CDN donde se encuentre localizado los archivos \*.css y \*.js del framework de Bootstrap 5.

Debemos modificar el archivo 'index.html' disponiendo los enlaces a los archivos respectivos y con eso ya tenemos acceso al framework de Bootstrap 5 en todo el proyecto:

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Proyecto043</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin="anonymous">

</head>

<body>

<app-root></app-root>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cDfL" crossorigin="anonymous"></script>

</body>

</html>

Modifiquemos la componente (app.component.html) que se ha creado por defecto y hagamos uso de las clases definidas en la librería de Bootstrap 5:

<div class="container">

<h1 class="display-4">Angular con Bootstrap</h1>

<p class="lead">Bootstrap 5 es un framework de CSS que facilita y estandariza el desarrollo de sitios web.

Se ha implementado pensando en adaptarse tanto a las pantallas de equipos de escritorio como a móviles y tablets.</p>

<p>Bootstrap 5 ha sido desarrollado y es mantenido por la comunidad de código abierto.</p>

<button type="button" class="btn btn-primary" data-bs-toggle="modal" data-bs-target="#dialogo1">Información</button>

<div class="modal fade" id="dialogo1">

<div class="modal-dialog">

<div class="modal-content">

<div class="modal-header">

<h5 class="modal-title">Información</h5>

<button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>

</div>

<!-- cuerpo del diálogo -->

<div class="modal-body">

<a href="https://getbootstrap.com/">Ingresar al sitio oficial</a><br>

<a href="https://tutorialesprogramacionya.com/bootstrap5ya">Tutorial de Bootstrap</a>

</div>

<!-- pie del diálogo -->

<div class="modal-footer">

<button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Cerrar</button>

</div>

</div>

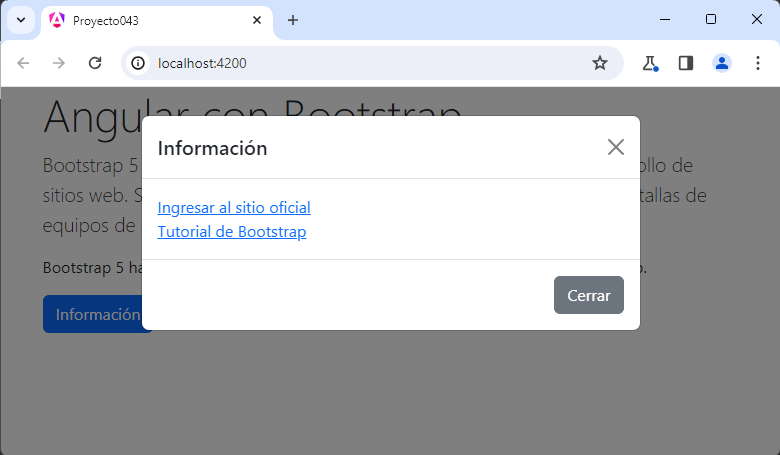
</div>

</div>

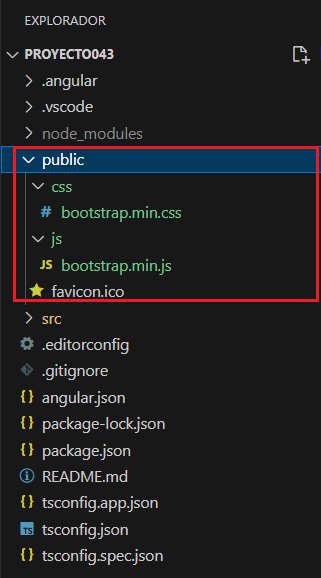
</div>

<router-outlet />

Si ejecutamos la aplicación podemos ver que tenemos disponible la librería \*.css de Bootstrap como también acceso a las funcionalidades de Javascript de Bootstrap en si mismo (de todos los elementos de Bootstrap que requieren Javascript):



1. Si queremos tener los archivos de Bootstrap en nuestro servidor y no en un CDN tenemos como primer alternativa descargarlos y copiar los archivos a la carpetas 'js' y 'css' que debemos crear en la carpeta 'public', luego tenemos los archivos en forma local:



Solo nos falta cambiar las url en el archivo 'index.html':

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Proyecto043</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

<link rel="stylesheet" href="/css/bootstrap.min.css">

</head>

<body>

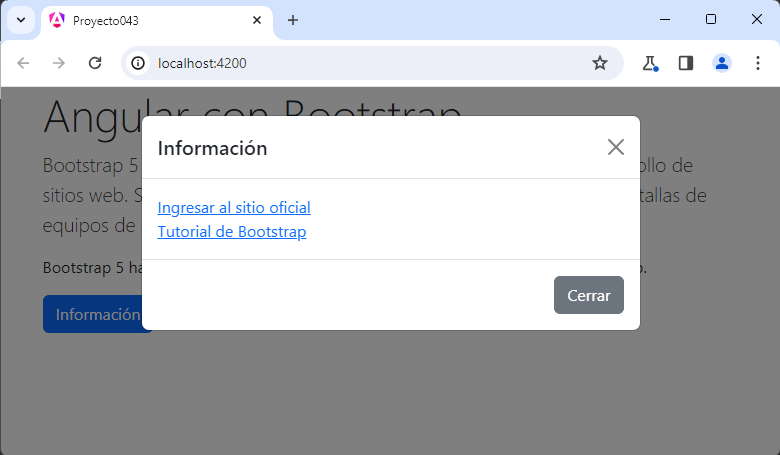
<app-root></app-root>

<script src="/js/bootstrap.min.js"></script>

</body>

</html>

Ya podemos ejecutar y tener el mismo resultado:



1. La tercer forma de instalar la última version de Bootstrap (actualmente la 5) es empleando el gestor de paquetes de Node.Js

Desde la línea de comandos, en la carpeta raiz de nuestro proyecto procedemos a instalar el paquete:

npm install bootstrap --save

La opción --save hace que npm incluya el paquete dentro de la sección de dependencies del archivo package.json en forma automática, lo que evita que tengamos que escribirlo en forma manual.

Si abrimos el archivo 'package.json' podemos ver que se han añadido las tres dependencias en nuestro proyecto:

{

"name": "proyecto043",

"version": "0.0.0",

"scripts": {

"ng": "ng",

"start": "ng serve",

"build": "ng build",

"watch": "ng build --watch --configuration development",

"test": "ng test"

},

"private": true,

"dependencies": {

"@angular/animations": "^19.0.0",

"@angular/common": "^19.0.0",

"@angular/compiler": "^19.0.0",

"@angular/core": "^19.0.0",

"@angular/forms": "^19.0.0",

"@angular/platform-browser": "^19.0.0",

"@angular/platform-browser-dynamic": "^19.0.0",

"@angular/router": "^19.0.0",

**"bootstrap": "^5.3.3",**

"rxjs": "~7.8.0",

"tslib": "^2.3.0",

"zone.js": "~0.15.0"

},

"devDependencies": {

"@angular-devkit/build-angular": "^19.0.2",

"@angular/cli": "^19.0.2",

"@angular/compiler-cli": "^19.0.0",

"@types/jasmine": "~5.1.0",

"jasmine-core": "~5.4.0",

"karma": "~6.4.0",

"karma-chrome-launcher": "~3.2.0",

"karma-coverage": "~2.2.0",

"karma-jasmine": "~5.1.0",

"karma-jasmine-html-reporter": "~2.1.0",

"typescript": "~5.6.2"

}

}

Como último paso para poder utilizar Bootstrap en nuestro proyecto, debemos modificar el archivo 'angular.json' donde debemos especificar el path de los 2 archivos (en la propiedad 'build', en 'styles' indicamos el path donde se encuenta el archivo \*.css y en 'script' el archivo respectivo):

{

"$schema": "./node\_modules/@angular/cli/lib/config/schema.json",

"version": 1,

"newProjectRoot": "projects",

"projects": {

"proyecto043": {

"projectType": "application",

"schematics": {},

"root": "",

"sourceRoot": "src",

"prefix": "app",

"architect": {

"build": {

"builder": "@angular-devkit/build-angular:application",

"options": {

"outputPath": "dist/proyecto043",

"index": "src/index.html",

"browser": "src/main.ts",

"polyfills": [

"zone.js"

],

"tsConfig": "tsconfig.app.json",

"assets": [

{

"glob": "\*\*/\*",

"input": "public"

}

],

"styles": [

**"node\_modules/bootstrap/dist/css/bootstrap.min.css",**

"src/styles.css"

],

"scripts": [

**"node\_modules/bootstrap/dist/js/bootstrap.min.js"**

]

},

"configurations": {

"production": {

"budgets": [

{

"type": "initial",

"maximumWarning": "500kB",

"maximumError": "1MB"

},

{

"type": "anyComponentStyle",

"maximumWarning": "4kB",

"maximumError": "8kB"

}

],

"outputHashing": "all"

},

"development": {

"optimization": false,

"extractLicenses": false,

"sourceMap": true

}

},

"defaultConfiguration": "production"

},

"serve": {

"builder": "@angular-devkit/build-angular:dev-server",

"configurations": {

"production": {

"buildTarget": "proyecto043:build:production"

},

"development": {

"buildTarget": "proyecto043:build:development"

}

},

"defaultConfiguration": "development"

},

"extract-i18n": {

"builder": "@angular-devkit/build-angular:extract-i18n"

},

"test": {

"builder": "@angular-devkit/build-angular:karma",

"options": {

"polyfills": [

"zone.js",

"zone.js/testing"

],

"tsConfig": "tsconfig.spec.json",

"assets": [

{

"glob": "\*\*/\*",

"input": "public"

}

],

"styles": [

"src/styles.css"

],

"scripts": []

}

}

}

}

}

}

Con este método no debemos modificar nada el archivo 'index.html':

<!doctype html>

<html lang="es">

<head>

<meta charset="utf-8">

<title>Proyecto043</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

</html>

Hemos presentado en este concepto las tres formas de integrar el Bootstrap original dentro de un proyecto Angular, el empleo de uno u otro depende de nuestra decisión. Veremos en los próximos conceptos algunos ejercicios empleando Angular y Bootstrap original.

66 - Angular y Bootstrap - creación de una componente: barrademenu

Como hemos tratado en todo el curso el corazón de Angular es dividir una aplicación en componentes y luego hacer que las mismas interactuen entre ellas, lograr crear componentes reutilizables y fáciles de mantener.

En este concepto y los próximos realizaremos una serie de componentes para practicar el desarrollo de componentes y utilizaremos en las mismas las herramientas que nos facilitan Bootstrap para las interfaces visuales.

Creación de una componente de menu de barra.

Crear una componente llamada MenudebarraComponent y cuyo selector debe llamarse 'boot-menudebarra'.  
Mediante 'property binding' se se debe poder comunicar un arreglo con las etiquetas que debe mostrar el menú, una segunda propiedad debe poder cargarse para indicar el color de fondo de la barra, los colores deben ser alguno de los valores clásicos propuestos por Bootstrap (bg-primary, bg-secondary, bg-success, bg-danger, bg-warning, bg-info, bg-light, bg-dark, bg-white)

Por otro lado desde la componente 'barrademenu' debe disparar un evento cuando el usuario selecciona una opción del menú.

* Crearemos primero el proyecto
* ng new proyecto044
* Instalamos el Bootstrap original con alguna de las tres formas vistas en el concepto anterior, mediante CDN o localizándolo en la capeta 'public' o mediante npm.
* Crearemos la componente de barra de menú e indicamos como prefijo para la nueva etiqueta la cadena 'boot' (la idea es recordarnos que esta componente utiliza la librería de Bootstrap):
* ng generate component barrademenu --prefix boot
* Modificamos el archivo 'barrademenu.component.ts':
* import { Component, Input, Output, EventEmitter } from '@angular/core';
* @Component({
* selector: 'boot-barrademenu',
* imports: [],
* templateUrl: './barrademenu.component.html',
* styleUrl: './barrademenu.component.css'
* })
* export class BarrademenuComponent {
* @Input() opciones!: string[];
* @Input() colorfondo!: string;
* @Output() presionopcion = new EventEmitter();
* presion(i: number): void {
* this.presionopcion.emit(i);
* }
* }

Definimos las dos propiedades que llegan datos a las componentes mediante el decorador @Input:

@Input() opciones!: string[];

@Input() colorfondo!: string;

Para definir el evento que dispara la componente:

@Output() presionopcion = new EventEmitter();

El método 'presion' se ejecuta cuando alguna de las opciones del menú de barra es presionada (informa a la componente padre el número de opción presionada):

presion(i: number): void {

this.presionopcion.emit(i);

}

* Modificamos el archivo 'barrademenu.component.html':
* <nav class="navbar navbar-expand-sm navbar-dark {{colorfondo}}">
* <ul class="navbar-nav">
* @for(opcion of opciones; track $index) {
* <li class="nav-item">
* <a class="nav-link" href="#" (click)="presion($index+1)">{{opcion}}</a>
* </li>
* }
* </ul>
* </nav>

Utilizamos clases propias de Bootstrap para definir el color de fondo que se remplaza mediante interpolación

<nav class="navbar navbar-expand-sm navbar-dark {{colorfondo}}">

Mediante un @for generamos todas las etiquetas 'li' con una etiqueta 'a' en su interior que muestra la opción y enlaza el evento 'click' pasando el número de opción seleccionada:

@for(opcion of opciones; track $index) {

<li class="nav-item">

<a class="nav-link" href="#" (click)="presion($index+1)">{{opcion}}</a>

</li>

}

* Ahora nos queda consumir la componente 'boot-barrademenu', esto lo haremos desde la componente principal de nuestra aplicación Angular.

Modificamos el archivo 'app.component.html':

<div class="container">

<boot-barrademenu [opciones]="opciones" colorfondo="bg-info" (presionopcion)="presion($event)"></boot-barrademenu>

</div>

<router-outlet />

Mediante enlace de propiedades enlazamos el arreglo a pasar a la propiedad 'opciones' de la componente:

[opciones]="opciones"

Para la propiedad 'colorfondo' directamente pasamos el string del color que queremos que aparezca la barra de menú:

colorfondo="bg-info"

Finalmente enlazamos el evento que dispara la componente 'presionopcion':

(presionopcion)="presion($event)

* Modificamos el archivo 'appcomponent.component.ts':
* import { Component } from '@angular/core';
* import { RouterOutlet } from '@angular/router';
* import { BarrademenuComponent } from './barrademenu/barrademenu.component';
* @Component({
* selector: 'app-root',
* imports: [RouterOutlet, BarrademenuComponent],
* templateUrl: './app.component.html',
* styleUrls: ['./app.component.css']
* })
* export class AppComponent {
* opciones = [
* "Opcion 1", "Opcion 2", "Opcion 3", "Opcion 4", "Opcion 5"
* ];
* presion(op: number) {
* alert("Se informa que se presiono la opcion " + op);
* }
* }

Definimos el arreglo con las opciones que tendrá la barra de menú:

opciones = [

"Opcion 1", "Opcion 2", "Opcion 3", "Opcion 4", "Opcion 5"

];

Definimos el método que recibe el número de opción seleccionado de la barra de menú:

presion(op: number) {

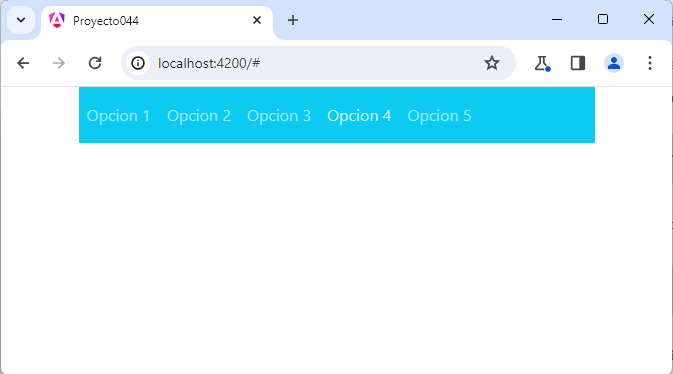
alert("Se informa que se presiono la opcion " + op);

}

Ahora ejecutemos la aplicación:

ng serve -o

Tenemos como resultado:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto044/).

67 - Angular y Bootstrap - creación de una componente: alerta

Creación de una componente de cuadro de alerta.

Crear una componente llamada AlertaComponent y cuyo selector debe llamarse 'boot-alerta'.  
Mediante dos propiedades se deben poder cargar el color y el mensaje que muestra el cuadro de alerta.

Los colores pueden ser alguno de los valores clásicos propuestos por Bootstrap (alert-primary, alert-secondary, alert-success, alert-danger, alert-warning, alert-info, alert-light, alert-dark, alert-white)

Probar la componente creada desde la componente principal de la aplicación.

* Crearemos primero el proyecto
* ng new proyecto045
* Instalamos el Bootstrap original con alguna de las tres formas vistas en conceptos anteriores, mediante CDN o localizándolo en la capeta 'public' o mediante npm.
* Crearemos la componente de alerta e indicamos como prefijo para la nueva etiqueta la cadena 'boot' (la idea es recordarnos que esta componente utiliza la librería de Bootstrap):
* ng generate component alerta --prefix boot
* Modificamos el archivo 'alerta.component.ts':
* import { Component, Input } from '@angular/core';
* @Component({
* selector: 'boot-alerta',
* imports: [],
* templateUrl: './alerta.component.html',
* styleUrl: './alerta.component.css'
* })
* export class AlertaComponent {
* @Input() color!: string;
* @Input() mensaje!: string;
* }

Definimos las dos propiedades que se inicializan con el color y el mensaje que debe mostrar la componente:

@Input() color!: string;

@Input() mensaje!: string;

* Modificamos el archivo 'alerta.component.html':
* <div class="alert {{color}} alert-dismissible fade show" role="alert">
* {{mensaje}}
* <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close">
* </button>
* </div>

Utilizamos clases propias de Bootstrap para definir el color

<div class="alert {{color}} alert-dismissible fade show" role="alert">

Por otro lado mediante interpolación se muestra el contenido de la propiedad 'mensaje'.

* Ahora nos queda consumir la componente 'boot-alerta', esto lo haremos desde la componente principal de nuestra aplicación Angular.

Modificamos el archivo 'app.component.html':

<div class="container">

<boot-alerta mensaje="Recuerde que las facturas vencen el 10 de cada mes" color="alert-warning"></boot-alerta>

</div>

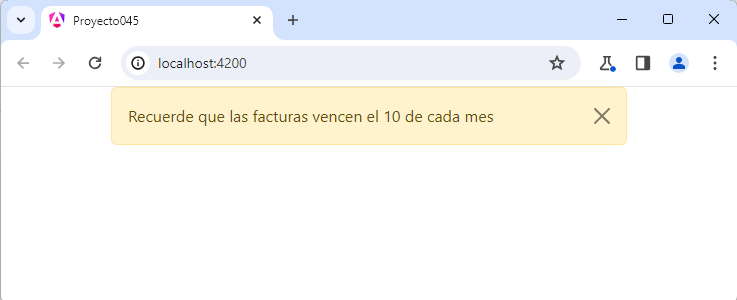
<router-outlet />

* En el archivo 'app.component.ts' importamos la componente:
* import { Component } from '@angular/core';
* import { RouterOutlet } from '@angular/router';
* import { AlertaComponent } from './alerta/alerta.component';
* @Component({
* selector: 'app-root',
* imports: [RouterOutlet, AlertaComponent],
* templateUrl: './app.component.html',
* styleUrls: ['./app.component.css']
* })
* export class AppComponent {
* title = 'proyecto045';
* }

Ahora ejecutemos la aplicación:

ng serve -o

Tenemos como resultado:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto045/).

68 - Angular y Bootstrap - creación de una componente: tabla

Creación de una componente de tabla de datos.

Crear una componente llamada TablaComponent y cuyo selector debe llamarse 'boot-tabla'.

Mediante una propiedad de tipo matriz de string poder pasar los datos que se deben mostrar en una tabla HTML. La primer fila de la matriz representa los títulos.

* Crearemos primero el proyecto
* ng new proyecto046
* Instalamos el Bootstrap original con alguna de las tres formas vistas en conceptos anteriores, mediante CDN o localizándolo en la capeta 'public' o mediante npm.
* Crearemos la componente de tabla e indicamos como prefijo para la nueva etiqueta la cadena 'boot' (la idea es recordarnos que esta componente utiliza la librería de Bootstrap):
* ng generate component tabla --prefix boot
* Modificamos el archivo 'tabla.component.ts':
* import { Component, Input } from '@angular/core';
* @Component({
* selector: 'boot-tabla',
* imports: [],
* templateUrl: './tabla.component.html',
* styleUrl: './tabla.component.css'
* })
* export class TablaComponent {
* @Input() datos: String[][]=[];
* }
* Modificamos el archivo 'tabla.component.html':
* <table class="table table-striped table-bordered">
* <thead>
* <tr>
* @for(titulo of datos[0]; track $index) {
* <th> {{ titulo }} </th>
* }
* </tr>
* </thead>
* <tbody>
* @for(fila of datos.slice(1); track fila) {
* <tr>
* @for(celda of fila;track celda) {
* <td>{{ celda }}</td>
* }
* </tr>
* }
* </tbody>
* </table>
* Ahora nos queda consumir la componente 'boot-tabla', esto lo haremos desde la componente principal de nuestra aplicación Angular.

Modificamos el archivo 'app.component.ts':

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

import { RouterOutlet } from '@angular/router';

import { TablaComponent } from './tabla/tabla.component';

@Component({

selector: 'app-root',

imports: [RouterOutlet, TablaComponent],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

da: string[][] = [

["Artículo", "Descripción", "Precio","Stock"],

["Artículo 1", "Descripción 1", "$10.00","30"],

["Artículo 2", "Descripción 2", "$20.00","33"],

["Artículo 3", "Descripción 3", "$15.00","72"],

["Artículo 4", "Descripción 4", "$25.00","91"],

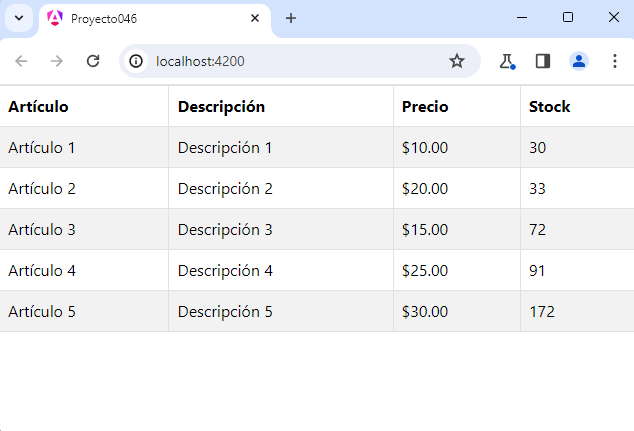
["Artículo 5", "Descripción 5", "$30.00","172"]

];

}

* Modificamos el archivo 'app.component.html':
* <boot-tabla [datos]="da"></boot-tabla>
* <router-outlet />
* Ahora ejecutemos la aplicación:
* ng serve -o

Tenemos como resultado:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto046/).

69 - Angular y Bootstrap - creación de una componente: tarjeta

Vamos a desarrollar una última componente haciendo uso del framework de Bootstrap original, para ver como integrarlo a Angular. Con esto podemos luego crear componentes visuales similares a las que propone Bootstrap pero con las funcionalidades de Angular.

Creación de una componente llamada tarjeta.

Crear una componente llamada TarjetaComponent y cuyo selector debe llamarse 'boot-tarjeta'.

Definir una interface llamada 'Tarjeta' que tenga como atributos la url de una imagen, el nombre de la persona y su mail.

* Crearemos primero el proyecto
* ng new proyecto047
* Instalamos el Bootstrap original con alguna de las tres formas vistas en conceptos anteriores, mediante CDN o localizandolo en la capeta 'public' o mediante npm.
* Crearemos la interface tarjeta:
* ng generate interface tarjeta

Y luego la codificamos:

export interface Tarjeta {

nombre: string,

mail: string,

foto: string

}

* Crearemos la componente 'tarjeta' e indicamos como prefijo para la nueva etiqueta la cadena 'boot' (la idea es recordarnos que esta componente utiliza la librería de Bootstrap):
* ng generate component tarjeta --prefix boot
* Modificamos el archivo 'tarjeta.component.ts':
* import { Component, Input } from '@angular/core';
* import { Tarjeta } from '../tarjeta';
* @Component({
* selector: 'boot-tarjeta',
* imports: [],
* templateUrl: './tarjeta.component.html',
* styleUrl: './tarjeta.component.css'
* })
* export class TarjetaComponent {
* @Input() datos: Tarjeta = { nombre: '', mail: '', foto: '' };
* }
* Modificamos el archivo 'tarjeta.component.html':
* <div class="card" style="width: 18rem;margin:1rem">
* <img src="{{datos.foto}}" class="card-img-top" alt="...">
* <div class="card-body">
* <h5 class="card-title">{{datos.nombre}}</h5>
* <p class="card-text">
* {{datos.mail}}
* </p>
* </div>
* </div>
* Ahora nos queda consumir la componente 'boot-tarjeta', esto lo haremos desde la componente principal de nuestra aplicación Angular.

Modificamos el archivo 'app.component.ts':

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

import { RouterOutlet } from '@angular/router';

import { TarjetaComponent } from './tarjeta/tarjeta.component';

import { Tarjeta } from './tarjeta';

@Component({

selector: 'app-root',

imports: [RouterOutlet, TarjetaComponent],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

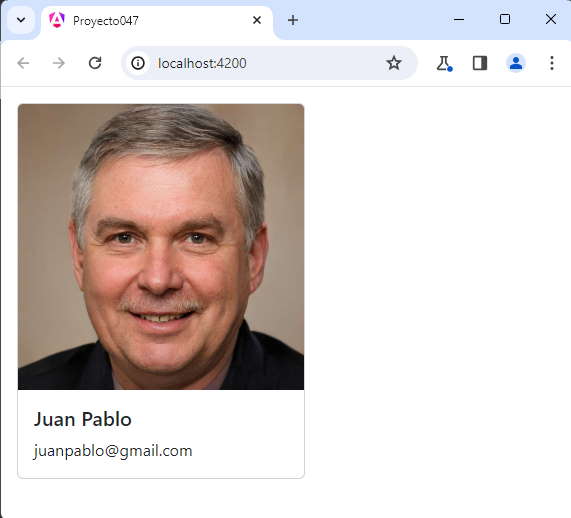
export class AppComponent {

tarjeta:Tarjeta={nombre:'Juan Pablo',mail:'juanpablo@gmail.com',foto:'https://www.ejerciciostutorialesya.com/angular/fotos/persona1.jpg'};

}

* Modificamos el archivo 'app.component.html':
* <boot-tarjeta [datos]="tarjeta"></boot-tarjeta>
* <router-outlet />
* Ahora ejecutemos la aplicación:
* ng serve -o

Tenemos como resultado:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto047/).

70 - Consumir una API pública utilizando una interface para definir su estructura y la clase HttpClient

Ya hemos visto en conceptos anteriores como podemos solicitar información a un servidor utilizando la clase HttpClient, pero no habíamos visto hasta ese momento la declaración de interfaces.

La declaración de la interface nos permite en tiempo de diseño identificar errores sobre nombre de atributos en los objetos.

Problema

La siguiente URL retorna un archivo JSON con los datos de distintos países:

https://www.ejerciciostutorialesya.com/cursojs/recuperarpaises.php

Como resultado tenemos un archivo similar a:

[

{

"nombre": "Argentina",

"capital": "Buenos Aires",

"idioma": "español",

"moneda": "peso",

"poblacion": 40000000,

"bandera": "https://www.ejerciciostutorialesya.com/cursojs/imagenes/argentina.png",

"limites": ["Chile", "Brasil", "Uruguay","Bolivia"]

},

{

"nombre": "Brasil",

"capital": "Brasilia",

"idioma": "portugués",

"moneda": "real",

"poblacion": 200000000,

"bandera": "https://www.ejerciciostutorialesya.com/cursojs/imagenes/brasil.png",

"limites": ["Argentina", "Chile", "Paraguay","Uruguay","Colombia","Venezuela","Bolivia","Peru","Ecuador","Guayana Francesa","Surinam","Guyana"]

},

{

"nombre": "Chile",

"capital": "Santiago",

"idioma": "español",

"moneda": "peso",

"poblacion": 18000000,

"bandera": "https://www.ejerciciostutorialesya.com/cursojs/imagenes/chile.png",

"limites": ["Argentina","Bolivia","Peru"]

}

]

Confeccionar una aplicación en Angular que recupere los datos del servidor y los muestre en una tabla HTML.

Vamos a codificar cada uno de los pasos para resolver el problema propuesto.

El resultado final debe ser similar a:

Tabla

El contenido generado por IA puede ser incorrecto.

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto048/).

* Crearemos primero el proyecto
* ng new proyecto048
* Debemos modificar el archivo 'app.config.ts' para poder hacer consultas a un servidor:
* import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
* import { provideRouter } from '@angular/router';
* import { routes } from './app.routes';
* import { provideHttpClient } from '@angular/common/http';
* import { withFetch } from '@angular/common/http';
* export const appConfig: ApplicationConfig = {
* providers: [provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes), provideHttpClient(withFetch())]
* };
* Creamos la interface con la estructura del archivo JSON:
* ng generate interface Pais

Declaramos la estructura de la interface:

export interface Pais {

nombre: string,

capital: string,

idioma: string,

moneda: string,

poblacion: number,

bandera: string,

limites: string[]

}

Como podemos ver los objetos contenidos del archivo JSON quedan mapeados con los nombres de las propiedades y sus tipos de datos. nombre es de tipo string, poblacion es numeric, limites es un array de string etc.

* Pasamos a crear ahora el servicio que va a emplear la clase HttpClient para comunicarse con el servidor y proceder a recuperar los datos en forma asíncrona:
* ng generate service paises

El código a implementar en el servicio es:

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

import { Pais } from './pais';

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

@Injectable({

providedIn: 'root'

})

export class PaisesService {

private urlApi = "https://www.ejerciciostutorialesya.com/cursojs/recuperarpaises.php"

constructor(private http: HttpClient) { }

retornar() {

return this.http.get<Pais[]>(this.urlApi);

}

}

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

Recordemos que la clase HttpClient es un servicio proporcionado por Angular para hacer solicitudes HTTP.

import { Pais } from './pais';

Es la interfaz que define la estructura de un objeto "Pais".

constructor(private http: HttpClient) { }

Al constructor se inyecta el servicio HttpClient.

retornar() {

return this.http.get<Pais[]>(this.urlApi);

}

Este método utiliza el servicio HttpClient para realizar una solicitud GET a la URL especificada (this.urlApi).  
El método devuelve un objeto de la clase Observable de tipo Pais[] (el próximo concepto ahondaremos el concepto de Observable). La notación indica que se espera una respuesta que sea un array de objetos que coincidan con la interfaz Pais.

Este método se utilizará en una componente donde se listarán los paises recuperados.

* Creamos la componente donde se mostrarán en una tabla HTML los paises:
* ng generate component ListadoPaises

Codificamos la clase 'ListadoPaisesComponent' contenida en el archivo 'listado-paises.component.ts'

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

import { PaisesService } from '../paises.service';

import { Pais } from '../pais';

@Component({

selector: 'app-listado-paises',

imports: [],

templateUrl: './listado-paises.component.html',

styleUrl: './listado-paises.component.css'

})

export class ListadoPaisesComponent {

paises: Pais[] = [];

constructor(private paisesService: PaisesService) {

this.recuperar()

}

recuperar() {

this.paisesService.retornar().subscribe((paises: Pais[]) => {

this.paises = paises

});

}

}

paises: Pais[] = [];

Definimos un arreglo inicialmente vacío, que almacenará los datos recuperados del servidor.

constructor(private paisesService: PaisesService) {

this.recuperar()

}

El constructor del componente recibe el servicio PaisesService como una dependencia e invoca el método recuperar automáticamente.

recuperar() {

this.paisesService.retornar().subscribe((paises: Pais[]) => {

this.paises = paises

});

}

El método recuperar utiliza el servicio PaisesService para recuperar la lista de países. Utiliza el método retornar() del servicio, que devuelve un observable (a partir del próximo concepto veremos en profundidad la clase Observable). El componente se suscribe al observable y actualiza la propiedad paises cuando se recibe la respuesta.

* Codificamos la clase la plantilla HTML 'listado-paises.component.html'
* <table>
* <thead>
* <tr>
* <th>Nombre</th>
* <th>Capita</th>
* <th>Idioma</th>
* <th>Moneda</th>
* <th>Poblacion</th>
* <th>Paises limitrofes</th>
* <th>Bandera</th>
* </tr>
* </thead>
* <tbody>
* @for(pais of paises;track pais.nombre) {
* <tr>
* <td>{{pais.nombre}}</td>
* <td>{{pais.capital}}</td>
* <td>{{pais.idioma}}</td>
* <td>{{pais.moneda}}</td>
* <td>{{pais.poblacion}}</td>
* <td>
* @for(limite of pais.limites;track limite) {
* {{limite}}<br>
* }
* </td>
* <td><img src="{{pais.bandera}}" alt="pais.nombre"></td>
* </tr>
* }
* </tbody>
* </table>
* Codificamos la hoja de estilo 'listado-paises.component.css'
* table {
* width: 100%;
* border-collapse: collapse;
* margin-bottom: 20px;
* }
* th, td {
* border: 1px solid #dddddd;
* padding: 8px;
* text-align: left;
* }
* tr:nth-child(even) {
* background-color: #f2f2f2;
* }
* th {
* background-color: #4CAF50;
* color: white;
* }
* Ahora modificamos la componente principal que crea Angular por defecto 'app.component.ts'
* import { Component } from '@angular/core';
* import { RouterOutlet } from '@angular/router';
* import { ListadoPaisesComponent } from './listado-paises/listado-paises.component';
* @Component({
* selector: 'app-root',
* imports: [RouterOutlet, ListadoPaisesComponent],
* templateUrl: './app.component.html',
* styleUrls: ['./app.component.css']
* })
* export class AppComponent {
* }

Finalmente modificamos la plantilla HTML de la componente principal que crea Angular por defecto 'app.component.html'

<app-listado-paises></app-listado-paises>

<router-outlet />

Ya podemos probar nuestra aplicación en forma local:

ng serve -o

Tenemos como resultado:

Tabla

El contenido generado por IA puede ser incorrecto.

Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto048/).

71 - Que utilizar en Angular: Promise u Observable

En Angular, tanto las promesas (Promise) como los observables (Observable) se utilizan para manejar operaciones asíncronas, pero tienen diferencias clave en su implementación y funcionalidad.

Similitudes entre Promesas y Observables:

* Manejo de operaciones asíncronas: Ambas promesas y observables son utilizados para manejar operaciones asíncronas en Angular, permitiendo que el código no se bloquee mientras espera la finalización de una operación.
* Manejo de errores: Tanto las promesas como los observables proporcionan mecanismos para manejar errores.

Diferencias entre Promesas y Observables:

* Promesas: Representan un único valor que será resuelto o rechazado.  
  Observables: Pueden representar una secuencia de valores que se emiten a lo largo del tiempo, permitiendo la transmisión de múltiples valores.
* Promesas: Una promesa no se puede cancelar.  
  Observables: Son cancelables, lo que significa que puedes cancelar una suscripción antes de que se complete la operación asíncrona.
* Promesas: Son más simples y adecuadas para operaciones asíncronas únicas.  
  Observables: Son más potentes y flexibles, especialmente cuando se trata de manejar secuencias de eventos a lo largo del tiempo. Son preferidos en situaciones más complejas y en el desarrollo de aplicaciones reactivas.

API Angular

La mayor parte de las librerías propuestas en angular utilizan por defecto los observables, se utilizan en servicios como HTTP, formularios reactivos, eventos del DOM, etc.

En resumen, mientras que las promesas son más simples y adecuadas para casos de uso más básicos, los observables proporcionan una funcionalidad más avanzada y son la elección preferida en el ecosistema Angular, especialmente en situaciones más complejas y en el desarrollo de aplicaciones más reactivas.

Problema

Vamos a desarrollar una aplicación que resuelva el mismo problema utilizando la clase Promise y la clase Observable.

Desarrollar una aplicación que permita al usuario seleccionar un archivo con formato HTML de su equipo y posteriormente muestre su contenido en la página. Resolver la actividad de la lectura en una Promise y luego con un Observable, utilizando la clase FileReader para la lectura del archivo. Debe realizar la lectura del archivo con formato HTML y si dicho archivo no tiene extensión HTML debe rechazar su lectura.

* Crearemos primero el proyecto
* ng new proyecto049
* Crear ahora el servicio donde vamos a hacer la lectura del archivo en forma asíncrona, empleando tanto promesas como observables:
* ng generate service archivo

El código a implementar en el servicio es:

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

import { Observable, Observer } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class ArchivoService {

constructor() { }

recuperarConPromesa(file: File) {

return new Promise<string>((resolve, reject) => {

if (!file.name.endsWith(".html")) {

reject("Extensión de archivo incorrecta")

} else {

const fileReader1 = new FileReader()

fileReader1.readAsText(file)

fileReader1.addEventListener("load", () => {

resolve(fileReader1.result!.toString())

})

}

})

}

recuperarConObservable(file: File): Observable<string> {

return new Observable((observer: Observer<string>) => {

if (!file.name.endsWith(".html")) {

observer.error("Extensión de archivo incorrecta");

} else {

const fileReader = new FileReader();

fileReader.readAsText(file);

fileReader.addEventListener("load", () => {

observer.next(fileReader.result!.toString());

observer.complete();

});

}

});

}

}

El método 'recuperarConPromesa' recibe un objeto File como parámetro, que se obtiene al seleccionar un archivo a través de un formulario en HTML.

Verifica si la extensión del archivo es ".html". Si no lo es, rechaza la promesa con un mensaje de error.

Si la extensión del archivo es correcta, crea un objeto FileReader y utiliza su método readAsText para leer el contenido del archivo como texto.

Se añade un evento "load" al FileReader, que se dispara cuando la operación de lectura es exitosa. En ese caso, se resuelve la promesa con el resultado.

El método 'recuperarConObservable' también toma un objeto File como parámetro. Al igual que en el método de promesa, verifica si la extensión del archivo es ".html". Si no lo es, emite un error a través del objeto Observer.

Si la extensión del archivo es correcta, crea un objeto FileReader y utiliza su método readAsText para leer el contenido del archivo como texto. Se añade un evento "load" al FileReader, que se dispara cuando la operación de lectura es exitosa. En ese caso, emite el resultado (el contenido del archivo convertido a cadena) a través de observer.next().

Finalmente, completa el observable con observer.complete(), indicando que la secuencia ha terminado.

* Modificamos la componente creada por defecto por Angular.

app.component.ts

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

import { RouterOutlet } from '@angular/router';

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

import { ArchivoService } from './archivo.service';

@Component({

selector: 'app-root',

imports: [RouterOutlet, FormsModule],

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

nombreArchivo = '';

resultado: any;

constructor(private archi: ArchivoService) { }

async recuperarConPromise(event: any) {

try {

this.resultado = await this.archi.recuperarConPromesa(event.target.files[0]);

} catch (error) {

this.resultado = error

}

}

recuperarConObservable(event: any) {

this.archi.recuperarConObservable(event.target.files[0]).subscribe({

next: (resultado) => {

this.resultado = resultado;

},

error: (error) => {

this.resultado = error;

}

});

}

}

nombreArchivo: Variable que almacena el nombre del archivo seleccionado.

resultado: Variable que almacena el resultado de la operación de recuperación de archivos.

El constructor recibe una instancia de ArchivoService a través de la inyección de dependencias.

El método recuperarConPromise: Método asincrónico que utiliza await para esperar a que la promesa retornada por recuperarConPromesa se resuelva. Actualiza this.resultado con el resultado de la operación o un mensaje de error si la promesa es rechazada.

El método recuperarConObservable: utiliza la función subscribe para suscribirse a un observable retornado por recuperarConObservable. El objeto pasado a subscribe define dos funciones de devolución de llamada: next para manejar el próximo valor emitido por el observable y error para manejar el error.

* La plantilla HTML de la componente también la modificamos.

app.component.html

<div>

<label for="archivohtml">Seleccione el archivo HTML:</label>

<ul>

<li>

Con Promise:<input type="file" [(ngModel)]="nombreArchivo" (change)="recuperarConPromise($event)">

</li>

<li>

Con Observable<input type="file" [(ngModel)]="nombreArchivo" (change)="recuperarConObservable($event)">

</li>

</ul>

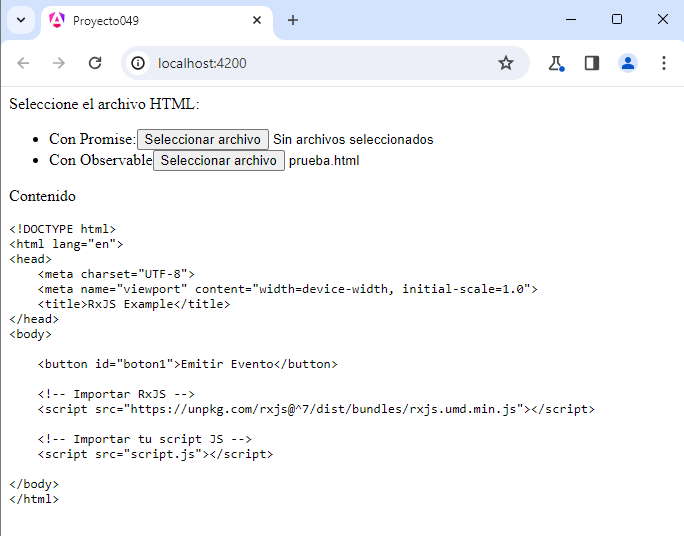
</div>

<p>Contenido</p>

<pre>{{resultado}}</pre>

<router-outlet />

La salida por pantalla luego de seleccionar un archivo HTML debe ser similar a:



Podemos probar esta aplicación en la web [aquí](https://ejerciciostutorialesya.com/angulardev/proyecto049/).