Apuntes de Angular

migueluisV

Abril 2024

Índice

Este documento se hizo con **Overleaf** y los ejemplos fueron desarrollados y probados con **Visual Studio Code**.

1 Introducción

1.1 ¿Qué es Angular?

Angular es utilizado para crear sitios web escalables con contenido dinámico que cambia con base en las interacciones del usuario.

Angular es un Framework y una plataforma de desarrollo, esto quiere decir que provee de un alto rango de herramientas que te ayudan a escribir, visualizar y desplegar un proyecto, al mismo tiempo que es una estructura sobre la cual puedes construir un proyecto y personalizarlo.

Angular se basa principalmente en HTML, CSS y Typescript.

1.2 Instalación y creación de proyectos

1.2.1 Instalación

Para instalar Angular en una máquina local, es necesario insertar el siguiente comando:

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

1.2.2 Creación de un proyecto

Para crear un nuevo proyecto de Angular, se escribe el siguiente comando, donde nombreProyecto es el nombre que le daremos al proyecto:

```
ng new nombreProyecto
```

1.2.3 Ejecución del proyecto

Para ejecutar el proyecto, se debe de dirigir a la carpeta donde este se haya creado y ejecutar el siguiente comando:

```
ng serve -o
```

1.3 Estructura de archivos del proyecto

La estructura de archivos de los proyectos Angular es la siguiente (depende también del IDE o editor de texto que se utilice, la estructura siguiente fue tomada de haber creado un proyecto con Visual Studio Code):

- src
  - app
    - app.component.css
    - app.component.html
    - app.component.spec.ts
    - app.component.ts

        * app.config.server.ts
        * app.config.ts
        * app.routes.ts
    - assets
        * .gitkeep
    - favicon.ico
    - index.html
    - main.server.ts
    - main.ts
    - styles.css
- .editorconfig
- .gitignore
- angular.json
- package.json
- package-lock.json
- README.md
- server.ts
- tsconfig.app.json
- tsconfig.json
- tsconfig.scep.json

Donde:

- index.html es la estructura HTML principal del proyecto.
- main.ts es el punto de entrada del proyecto cuando este es ejecutado.
- styles.css es la hoja de estilos CSS principal del proyecto.
- app.component.ts y app.component.html representan el componente raíz del proyecto.

2 Conceptos básicos

2.1 Componentes

Los componentes son bloques que representan un objeto de interfaz en un proyecto Angular (así como los componentes de React), estos componentes son piezas reutilizables de código.

Un componente está compuesto de tres partes: **HTML** (estructura), **Typescript** (lógica) y **CSS** (estilos).

El componente lógico raíz de todo proyecto es el nombrado app.component.ts en la carpeta src/app, este componente tiene un decorador el cual especifica cual es su estructura HTML y su hoja de estilos:

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html', // estructura html
  styleUrl: './app.component.css' // hoja de estilos
})
export class AppComponent {
  title = 'prueba1';
}
```

Habíamos dicho anteriormente que **index.html** es la estructura principal del proyecto, veamos su contenido:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Prueba1</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>
```

Vemos que lo único que está en el body es una etiqueta llamada <app-root>, la cual no es original de HTML. Este tipo de etiquetas representan a los componentes de Angular, un componente tiene su nombre el cual es traducido a una etiqueta HTML y se puede insertar dentro del body de index.html e incluso insertar componentes dentro de otros componentes. ¿Cómo funciona esto?

Fijémonos en el único componente existente en un proyecto recién creado: app.component. Este componente es la raíz del proyecto, es lo que se muestra cuando se despliega el proyecto en un servidor, este componente en sí está constituido por tres partes:

- su estructura: la estructura HTML, la cual la encontramos en app.component.html.
- su lógica: la lógica y propiedades del componente, lo encontramos en app.component.ts.
- su estilo: la hoja de estilos, la cual la encontramos en app.component.css.

Esto quiere decir que un componente tiene diversas partes, en app.component.ts, en su decorador @Component encontramos la propiedad **selector**, la cual proporciona al componente (y a sus partes en conjunto) de un nombre para utilizarlo como una etiqueta HTML para ser insertada en alguna otra parte del proyecto, es así que podemos crear diversos componentes con su estructura, lógica y estilos para luego nombrarlos en @Component y utilizarlos en nuestro proyecto.

Nota: información adicional para este documento extraída de este link.

2.1.1 Creación de un componente

Se utiliza el siguiente comando, sin embargo, podemos añadir una instrucción al final que permitirá que el componente trabaje de distinta manera:

ng generate component nombreComponente

ng generate component nombreComponente –inline-template

Donde nombreComponente es el nombre que le daremos al componente, *–inline-template* te permite insertar etiquetas HTML cortas directamente al componente, sin esta indicación, al momento de la creación del componente, se crearían cuatro archivos alojados en una carpeta dentro de app con el nombre que se le fue dado al componente:

- *nombreComponente.component.ts*,
- *nombreComponente.component.html*,
- *nombreComponente.component.css* y
- *nombreComponente.component.spec.ts*

Los mismo que existen para el componente raíz (app.component). En nombreComponente.component.ts, en el decorador @Component, el valor de la propiedad *selector* es "app-nombreComponente", Angular agrega el prefijo "app-" a los componentes creados.

Para fines del siguiente código, sustituiremos nombreComponente por "footer". Este es el aspecto que tiene el nuevo componente en nuestro directorio y su código generado:

Archivos del proyecto

- src
  - app
    - footer
      - footer.component.css
      - footer.component.html
      - footer.component.ts
      - footer.component.spec.ts

    - app.component.css
    - app.component.html
    - app.component.spec.ts
    - app.component.ts
    - app.config.server.ts
    - app.config.ts
    - app.routes.ts

- ...

nombreComponente.component.ts

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

@Component({
  selector: 'app-footer',
  standalone: true,
  imports: [],
  templateUrl: './footer.component.html',
  styleUrl: './footer.component.css'
})
export class FooterComponent {}
```

nombreComponente.component.html

```html
<p>footer works!</p>
```

nombreComponente.component.css

(vacio)

nombreComponente.component.spec.ts

```ts
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { FooterComponent } from './footer.component';

describe('FooterComponent', () => {
  let component: FooterComponent;
  let fixture: ComponentFixture<FooterComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [FooterComponent]
    })
    .compileComponents();

    fixture = TestBed.createComponent(FooterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
```

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

2.1.2 Uso de componentes en otros componentes

Ya que tenemos creado nuestro componente nuevo, debemos importarlo a *app.component.ts* para que lo tenga disponible para su uso y así poder utilizarlo en *app.component.html* para su visualización; recuerde que el selector del componente Footer es "app-footer". Las modificaciones que haremos son las siguientes:

app.component.ts

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { FooterComponent } from './footer/footer.component'; // se importa el componente.

@Component({
  selector: 'app-root',
  standalone: true,
  //imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',
  imports: [FooterComponent] // el componente se pasa a un arreglo de importaciones para
      app.component.
})
export class AppComponent {
  title = 'prueba1';
}
```

app.component.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h4>hola</h4>
  <app-footer /> <!-- se utiliza app-footer -->
</body>
</html>
```

Nota: información adicional para este documento extraída de este link.

2.2 Plantillas

La plantilla de un componente también se le conoce como su estructura HTML (vista anteriorment). Se pueden utilizar variables que aparecen en la parte lógica de un componente en su estructura HTML para mostrar información o cambiar la misma desde la estructura.

Supongamos que tenemos una variable en el componente raíz de un proyecto llamada *name*:

```
user = 'James'
```

Si queremos mostrar este valor en la estructura del componente, utilizamos la sintaxis llamada **interpolación**:

```
<h1>Welcome, {{ name }}!</h1>
```

De esta manera, cuando el valor de *name* cambie, su cambio se verá reflejado en la interfaz. Como en React, también podemos usar la interpolación para asignar valores a los atributos de una etiqueta HTML:

```
<img src="{{imageURL}}" alt="{{altText}}" />
```

Del mismo modo, se pueden implementar llamadas a funciones o expresiones simples:

```
<h1>Welcome, {{ firstname + ' ' + lastname }}!</h1>
```

2.3 Vinculación

Vimos anteriormente que podemos mostrar valores de la lógica del componente a su estructura, esto corresponde con el concepto **Binding** (vinculación o encuadernación), el cual se refiere a la conexión en tiempo real entre una clase y su interfaz.

Angular permite vincular un atributo de una etiqueta HTML a una propiedad de su componente, logrando así que cuando el valor de la propiedad cambie, se refleje en la estructura, esto es útil en cuanto a las imágenes, se utilizan los corchetes ([]) para lograr esta vinculación:
app.component.ts

```
imageURL = 'tree.jpg'
```

app.component.html

```
<img [src] = 'imageURL'>
```

Se debe tener precaución cuando se utilicen los corchetes para encerrar un atributo de una etiqueta, el nombre a encerrar debe ser el mismo que el del atributo, o agregarle mayúsculas en ciertas ubicaciones, no se debe poner el nombre de la propiedad del componente. Se muestra un ejemplo donde si está bien y otro donde no:

Bien

```
<tr><td [colSpan]="columnsCount">Some text</td></tr>
<tr><td [colspan]="columnsCount">Some text</td></tr>
```

Mal

```
<tr><td [columnsCount]="columnsCount">Some text</td></tr>
```

También podemos lograr una vinculación usando clases y estilos de CSS, esto mediante la palabra reservada *class.* y *style.*:

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
})
export class AppComponent {
  isHighlighted = true;
  styleBackgroundColor = "#ff0000"
}
```

app.component.html

```
<p [class.highlight]="isHighlighted">some text</p>
<p [style.background-color]="styleBackgroundColor">siiiiii</p>
```

Se pueden utilizar múltiples clases en una vinculación usando solo la palabra reservada *class*, siempre y cuando:

- se pase como valor un arreglo con nombres de clases tipo string.
- una dupla donde una parte sea el nombre de la clase y la otra su valor (true o false).

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
})
export class AppComponent {
  myClasses = ['highlight', 'uppercase']
}
```

app.component.html

```
<p [class]="myClasses">some text</p>
```

Nota: información adicional para este documento extraída de este link.

2.4 Eventos

Continuando con el tema de la vinculación, Angular te permite vincular eventos con métodos de un componente, volviendolo responsivo a clics y otras cosas que puedan ocurrir. Un ejemplo muy básico es el siguiente:

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

Se encierra entre paréntesis el nombre del evento y se le asigna el nombre de la función entre dobles comillas.

2.5 Flujos de control

2.5.1 for

Por lo general, se suele repetir código el cual varia solamente por su contenido, por ejemplo: publicaciones en un blog, productos en una tienda en línea, imágenes en una galería. Para esto, Angular tiene una sintaxis especial para lidiar con el problema, el cual es @for:

```
@for (item of items; track item.id) {
   {{ item.name }}
}
```

La palabra reservada track se utiliza en esta sintaxis porque representa el id o índice del arreglo o colección de datos que navega, por lo que es único y es obligatorio ponerlo. En caso de que no importe mucho si los valores son únicos en el arreglo o colección que recorrerá, se puede poner la palabra *track* y el nombre de variable $index. Veamos un ejemplo para que se entienda mejor:

app.component.ts[style=htmlcssjs]

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
})
export class AppComponent {
  products = ['apple', 'orange', 'banana'];
}
```

app.component.html

```
@for (item of products; track $index) {
   <div>{{ item }}</div>
}
```

Repetirá el bloque *div* el número de ítems que tenga el arreglo *products*.

2.5.2 if

También se pueden poner ciertos valores siempre y cuando se cumpla alguna condición:

```
@if (loggedIn) {
   <div>Welcome!</div>
}
```

3 Formularios

3.1 Template-driven

Una de las formas de crear formularios en Angular es mediante los tipos de formularios **template-driven**, los cuales son creados y mantenidos en la estructura HTML (plantilla). Para crear uno, primero se debe importar la dependencia FormsModule al componente donde tendremos nuestro formulario:

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
  imports: [FormsModule],
})
```

Un código sencillo es el siguiente:

```
<label>Name:
  <input type="text" [(ngModel)]="name" />
</label>
```

Vemos que se utiliza un atributo o directiva **[(ngModel)]**, lo que hace esta directiva es que vincula (bind) el valor recibido en el control del formulario a una propiedad de la lógica o clase del componente (el app.component.ts por ejemplo).

Otra característica de esta directiva es que es bidireccional, es decir que el valor que se ingrese en el control del formulario afecta a la propiedad en la parte lógica del componente, del mismo modo que cambiar el valor de la propiedad en la lógica se ve reflejado en el control del formulario con la que está vinculada.

En caso de tener que manejar el evento **submit** de un formulario, se puede utilizar la directiva [(ngSubmit)] para ejecutar un método al enviar la información de un formulario. Si se utiliza el contenedor <form>, cada input o control de este debe tener obligatoriamente un atributo *name*.

```
<form (ngSubmit)="showName()">
  <input type="text" [(ngModel)]="name" name="name" />
  <input type="submit" value="Submit" />
</form>
```

3.1.1 Validación

Se pueden utilizar algunos atributos de etiquetas HTML para validar información sin ningún tipo de sintaxis especial, como *required*. Otra forma de validar la información de un formulario tipo **template-driven** es mediante la directiva **ngForm**, lo que hace esta directiva es que te da acceso al estado completo del formulario, la forma de utilizarla es la siguiente:

```
<form (ngSubmit)="showName()" #myForm="ngForm">
```

Primero le damos un nombre al formulario con el prefijo # y a esto le asignamos la directiva. De esta manera, podemos acceder al estado del formulario y realizar algunas validaciones antes de enviar información, por ejemplo:

```
<form (ngSubmit)="showName()" #myForm="ngForm">
  <input type="text" [(ngModel)]="name" name="name" required />
  <input type="submit" value="Submit" [disabled]="!myForm.form.valid" />
</form>
```

Vemos que se le da el nombre "myForm" a un formulario y se le asigna la directiva ngForm, su primer control o input es una caja de texto cualquiera, pero tiene la particularidad de que es requerido que sea llenada para enviar la información del formulario, entonces, el estado del formulario no es "completado" o "válido" hasta que se llene ese único control (en caso de que haya más controles con el atributo required, se deben llenar todo estos para que el estado sea "válido"), es por ello que el botón (el cual se encarga de hacer la operación de *submit*) tiene el atributo [disabled] que depende de que el estado del form (myForm.form) sea válido (myForm.form.valid).

Angular proporciona estilos especiales para los estados de los formularios para poder personalizarlos en caso de ser requerido:

```
input.ng-valid {
  background-color: #79ba6a;
}
input.ng-invalid {
  background-color: #f58c84;
}
```

3.2 Reactive Forms

La otra forma de crear formularios es mediante los tipos de formularios **Reactive forms**. Para crear uno, primero se debe importar la dependencia ReactiveFormsModule al componente donde tendremos nuestro formulario:

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
  imports: [ReactiveFormsModule],
})
```

Las dos diferencias principales entre este tipo y el primero es que este se crea y mantiene desde la lógica del componente y el primero desde su estructura, además que el primero está más enfocado en formularios pequeños y sencillos. La forma de instanciar este tipo desde la lógica es la siguiente:

```
import { FormControl } from '@angular/forms';
...
export class AppComponent {
  // Sin valor predeterminado.
```

```
  name = new FormControl('');

  // Con valor predeterminado
  name = new FormControl('Juan');
}
```

Puedes asignar un valor predeterminado al control.Después, tienes que asociar este control de formulario a un control en la estructura HTML:

```
<input type="text" [formControl]="name" name="name" />
```

La directiva [**formControl**] se usa para vincular la propiedad de la parte lógica con el control de la estructura. Se utiliza el siguiente código para acceder a la información ingresada del control:

```
<p>{{ name.value }}</p>
```

A diferencia del tipo anterior en cuanto al evento submit, cambia un poco la sintaxis de la directiva, en lugar de utilizar [(ngSubmit)] se utiliza simplemente (submit).

Este es el código completo:

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
  imports: [ReactiveFormsModule],
})

export class AppComponent {
  name = new FormControl('');

  login() {
    alert(this.name.value);
  }
}
```

app.component.html

```
<form (ngSubmit)="login()">
  <p><input type="text" [formControl]="name" /></p>
  <p><input type="submit" value="Submit" /></p>
</form>
```

3.3 Agrupación de controles

Para tener mayor orden en el número de controles de un formulario, se puede importar la dependencia FormGroup para agrupar controles:

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

Se crea el grupo nuevo en la parte lógica del componente:

```
loginForm = new FormGroup({
    username: new FormControl(''),
    password: new FormControl(''),
});
```

El nuevo grupo es llamado **loginForm** y sus controles son **username** y **password**. Esta declaración nos da como resultado una colección de datos dupla. Ahora debemos vincular este grupo con la estructura del componente:

```
<form [formGroup]="loginForm">
  <p><input type="text" formControlName="username" /></p>
  <p><input type="text" formControlName="password" /></p>
</form>
```

Así es como se accede a la información registrada de los controles de un grupo:

```
{{ loginForm.value.username }}
```

No cambia mucho la situación al momento de enviar la información del formulario:
app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
  imports: [ReactiveFormsModule],
})
export class AppComponent {
  loginForm = new FormGroup({
    username: new FormControl(''),
    password: new FormControl(''),
  });

  login() {
    alert(
      this.loginForm.value.username + ' | ' + this.loginForm.value.password
    );
  }
}
```

app.component.html

```
<form [formGroup]="loginForm" (ngSubmit)="login()">
  <p><input type="text" formControlName="username" /></p>
  <p><input type="text" formControlName="password" /></p>
  <p><input type="submit" value="Submit" /></p>
</form>
```

3.4 Validación mediante Validators

Otra forma de validar la información de los controles de un Reactive Form es mediante la dependencia Validators, la cual agrega características especiales y ahorra algo de código en la estructura del componente. Se importa de la siguiente manera:

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

Similar al estado del formulario, se puede revisar el estado de un grupo con una sintaxis similar:

```
<form [formGroup]="loginForm" (ngSubmit)="login()">
  <p><input type="text" formControlName="username" /></p>
  <p><input type="text" formControlName="password" /></p>
  <p><input type="submit" value="Submit" [disabled]="!loginForm.valid" /></p>
</form>
```

Note que la sintaxis ya no incluye la palabra *form* en medio del nombre del grupo y el atributo *valid*. En styles.css también puede incorporar los estilos **ng-valid** y **ng-invalid** a un grupo de controles.

Un botón también puede resetear los valores de un grupo con el método **reset()**:

```
<form [formGroup]="myForm">
   ...
   <input type="button" value="Reset" (click)="myForm.reset()" />
</form>
```

4 Routing

Para pasar de una pantalla a otra (o de un componente a otro), hay varios pasos a seguir: primero se importa la dependencia **RouterModule** de *@angular/router*.

```
import { RouterModule } from "@angular/router";
```

Luego, se agrega como una dependencia del componente que manejará el ruteo:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  standalone: true,
  imports: [RouterModule],
})
```

Para comprender como funcionará el ruteo que haremos, plantearemos el siguiente caso: tenemos una app que tiene dos pantallas o componentes, el home y una página de contactos, ambos tienen el mismo header, el cual tiene los dos link que navegan a las dos páginas descritas anteriormente. Este es el aspecto que tendría nuestro **app.component.ts**, quien es quien se encargará del ruteo:

```
import { Component } from '@angular/core';
import { HeaderComponent } from '../header/header.component';
import { RouterModule } from '@angular/router';

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

app.component.html

```
<app-header>
<router-outlet>
```

app.header.ts

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

@Component({
  selector: 'app-header',
  standalone: true,
  templateUrl: './header.component.html',
  styleUrl: './header.component.css',
  imports: [RouterModule],
})
export class HeaderComponent {}
```

app.header.html

```
<h1>My Header</h1>
<nav>
  <a routerLink="/home">Home</a>
  <a routerLink="/contacts">Contacts</a>
</nav>
```

De primeras, si hacemos estos cambios en un proyecto, lanzará un error ya que todavía no existe 'router-outlet'. Nos dirigiremos al archivo **main.ts** para crear el arreglo de rutas que utilizará el proyecto: **main.ts**

```
import 'zone.js';
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
import { provideRouter, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ContactsComponent } from './contacts/contacts.component';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'contacts', component: ContactsComponent },
];

bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes)],
});
```

Se importan los componentes **Home** y **Contacts** para utilizarlos más adelante, se importa también la librería **Routes** y **provideRouter**, el primero sirve para crear el arreglo de rutas y el segundo es para vincular un componente a una ruta creada, en *bootstrapApplication* es donde se pasan las rutas y vinculaciones al componente raíz. Home y Contacts tienen el aspecto normal de un componente recién creado sin importaciones referentes al ruteo.

4.1 Ruta predeterminada

Podemos definir un componente por defecto el cual será cargado cuando se acceda al proyecto, simplemente creamos una ruta con unas comillas simples vacías y el componente a donde queremos que vaya:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'home', component: HomeComponent },
  { path: 'contacts', component: ContactsComponent },
];
```

4.2 Rutas para errores

En caso de que se acceda a una Url que ya no existe, Angular puede redireccionar al usuario a una página de error, esto mediante las rutas comodín (widlcard routes) utilizando dos asteriscos en una de las rutas de nuestro proyecto:

```
const routes: Routes = [
  { path: '', title: 'Home', component: HomeComponent },
  { path: 'home', title: 'Home', component: HomeComponent },
  { path: 'contacts', title: 'Contacts', component: ContactsComponent },
  { path: '**', title: 'Error', component: UnknownComponent },
];
```

Es importante que esta ruta esté al final de todas, ya que lo que hace Angular es recorrer el arreglo de rutas desde el principio al final cuando un usuario hace una solicitud de acceso a nuestra app mediante una Url, si se recorrieron todas las rutas y no encontró la que el usuario solicitó, mostrará la que tenemos programada como la última que redirecciona a la página de error

4.3 Título de página

Cuando estamos en alguna página, la pestaña del buscador nos muestra un título, podemos modificarlo en Angular agregando el atributo **title** a las rutas que creemos para nuestro proyecto:

```
const routes: Routes = [
  { path: '', title: 'Home', component: HomeComponent },
  { path: 'home', title: 'Home', component: HomeComponent },
  { path: 'contacts', title: 'Contacts', component: ContactsComponent },
];
```

4.4 Uso de métodos para navegación

Podemos crear enlaces o botones que redirijan al usuario cuando este les de clic. Pondremos de ejemplo que Home dirija a Contacts y viceversa: **contacts.component.html**

```
import { Component, inject } from "@angular/core";
import { Inject } from "@angular/core";
import { Router } from "@angular/router";

@Component({
    selector: 'app-contacts',
    templateUrl: './contacts.component.html',
    standalone: true,
    imports: [],
})

export class ContactsComponent {
    router = inject(Router);

    navigate() {
        this.router.navigateByUrl('/home');
    }
}
```

contacts.component.ts

```
<h1>Contacts</h1>

<button (click)="navigate()">Home</button>
```

Simplemente sustituya la palabra "contacts" por home en el selector, templateUrl y el nombre de la clase pra tener el código del componente Home. Tome en cuenta que, para que este ejemplo funcione, debe complementarlo con la creación de las rutas y configuración de las mismas.

Nota: este ejemplo es de una versión de Angular anterior a la que está vigente al momento de la redacción de este documento. La forma de hacer routing básico con la versión que tengo descargada en este momento (17.3.3) es:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  standalone: true,
  imports: [RouterModule],
})

export class AppComponent {}
```

app.component.html

```
<router-outlet />
```

app.config.ts

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)]
};
```

app.routes.ts

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { ContactsComponent } from './contacts/contacts.component';

export const routes: Routes = [
    { path: '', title: 'Home', component: HomeComponent },
    { path: 'home', title: 'Home', component: HomeComponent },
    { path: 'contacts', title: 'Contacts', component: ContactsComponent }
];
```

main.ts

```
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));
```

Donde:

- **app.component.ts** y **app.component.html** si reciben modificaciones.
- **app.config.ts** no recibe modificaciones, se encarga de hacer el routing por nosotros.
- **app.routes.ts** recibe modificaciones, aquí se agregan las rutas que usará *app.config.ts* y *main.ts*.
- **main.ts** no recibe modificaciones.
- fuera de estos componentes, se deben agregar otros para crear las rutas y, al final de cuentas, el ruteo de la app.

4.5 Pasar parámetros entre Urls

Una función muy útil al momento de trabajar con el routing es poder pasar parámetros entre páginas para visualizar la información a detalle de un elemento, esta tarea es algo compleja de llevar a cabo pero la explicaremos paso a paso. Este ejemplo utiliza un componente Home y Notes, el primero muestra una lista de notas (que obtenemos de un archivo *.ts* en nuestro proyecto) y si le damos clic a alguna de ellas, se nos redirecciona a otro componente donde vemos más información de esa nota seleccionada.

Vamos a utilizar la última arquitectura del proyecto mencionada en este documento, estos son los componentes:

app.component.ts

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

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

app.component.html

```
<router-outlet></router-outlet>
```

No hay nada nuevo con estos dos archivos.

notes.component.ts

```
import { Component, inject } from "@angular/core";
import { ActivatedRoute, RouterModule, Router } from "@angular/router";
import { NOTES } from "../notes";

@Component({
    selector: 'app-notes',
    templateUrl: './notes.component.html',
    standalone: true,
    imports: [RouterModule],
})

export class NotesComponent {
    router = inject(Router);
    activatedRouter = inject(ActivatedRoute)
    id = Number(this.activatedRouter.snapshot.paramMap.get('id'));
    note = NOTES.find((x) => x.id === this.id)

    navigate() {
        this.router.navigateByUrl('/home');
    }
}
```

Vemos que el componente sigue importanto RouterModule, se sigue usando una variable que recibe la inyección de un objeto Router (para navegar mediante Urls), pero aquí comienza el primer cambio. Se importa la dependencia ActivatedRouter, la cual nos permite obtener valores de parámetros de una Url, se inyecta a una variable un objeto de esta dependencia y con el podemos acceder a los valores de los parámetros, en este caso, se le asigna a la variable **id** este valor de tipo numérico, con él, se busca en el arreglo NOTES (que se verá más adelante).

notes.component.html

```
<h1>Note details</h1>

@if (note) {
    <span>{{ note.title }}</span><br>
    <span>{{ note.id }}</span><br>
    <span>{{ note.text }}</span><br>
}

<button (click)="navigate()">Go back</button>
```

La plantilla del componennte simplemente verifica si **note** tiene algún valor, en caso de que si, muestra la información del mismo.

home.component.ts

```
import { Component, inject } from "@angular/core";
import { Inject } from "@angular/core";
import { RouterModule } from "@angular/router";
import { NOTES } from "../notes";

@Component({
    selector: 'app-home',
    templateUrl: './home.component.html',
```

```
    standalone: true,
    imports: [RouterModule],
})

export class HomeComponent {
    notes = NOTES;
}
```

home.component.html

```
<h1>Home</h1>

@for (note of notes; track $index) {
    <div>
        <button [routerLink]="['/notes', note.id]">
            <span>{{ note.id }}</span>
            <span>{{ note.text }}</span>
        </button>
    </div>
}
```

No hay mucha novedad en estos archivos, simplemente se importa el archivo de notas que estamos utilizando y se muestra en la plantilla del componente.

notes.ts

```
export interface Note {
    id: number;
    title: string;
    text: string;
  }
  export const NOTES: Note[] = [
    {
      id: 1,
      title: 'Lorem ipsum',
      text: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
          incididunt ut labore et dolore magna aliqua.',
    },
    {
      id: 2,
      title: 'Shakespeare',
      text: 'To be, or not to be: that is the question.',
    },
  ];
```

Este es el archivo de notas, es una interfaz con sus atributos y un arreglo de dos elementos, el cual se utiliza por los archivos anteriores.

app.routes.ts

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { NotesComponent } from './notes/notes.component';

export const routes: Routes = [
    {path: '', title: 'Home', component: HomeComponent},
    {path: 'home', title: 'Home', component: HomeComponent},
```

```
    {path: 'notes/:id', title: 'Contacts', component: NotesComponent}
];
```

La gran diferencia en este archivo es que en la ruta del componente Notes es que se le agrega una diagonal, dos puntos y el nombre del parámetro (notes/:id), de esta manera, **Routes** y **RouterModule** saben que la ruta tiene parámetros y la permite manejar en con los componentes.