ANGULAR

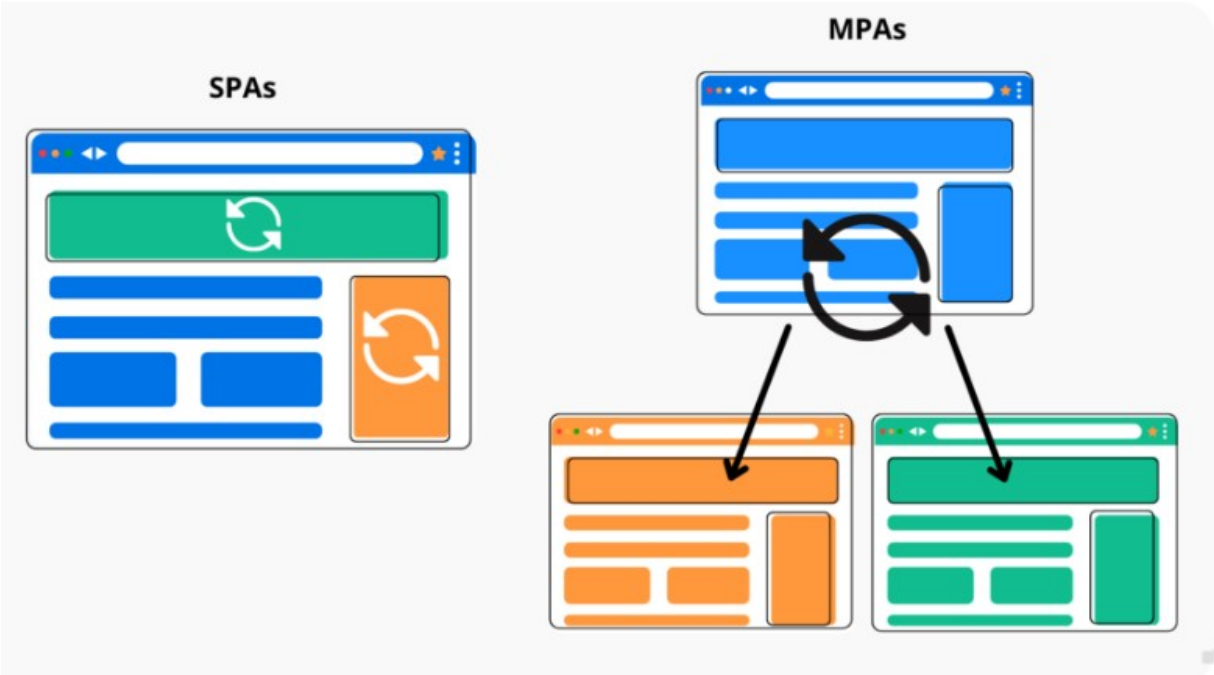
CONTENIDOS:

CONTENIDOS:

11. **Autenticación**
12. Usar BootStrap, jQuery, popper y FontAwesome con Angular.
13. Despliegue de aplicaciones a producción.

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

SPA vs MPA



Introducción

1.- Angular es un framework de desarrollo web creado por Google.

Fue lanzado en 2016 como una evolución de AngularJS, para aplicaciones modernas y escalables.

2.- Basado en TypeScript.

TypeScript es un superconjunto de JavaScript que añade tipado estático y funcionalidades avanzadas.

3.- Permite crear aplicaciones web dinámicas y de alto rendimiento.

Gracias a su arquitectura modular y herramientas integradas, optimiza el desarrollo web.

4.- MVVM (Model-View-ViewModel) como patrón de diseño.

Separa la lógica de la vista y facilita la reutilización del código y el mantenimiento.

Principales características de Angular

- **Componentes:** Es una porción de código que es posible reutilizar en otros proyectos de Angular.
- **Módulos:** Organización del código en secciones reutilizables.
- **Plantillas:** HTML con extensiones de Angular.
- **Inyección de dependencias:** Gestión eficiente de servicios.
- **Herramientas CLI:** Facilita el desarrollo

Beneficios de Angular

- Desarrollo modular y escalable.
- Comunidad amplia y soporte oficial de Google.
- Compatible con múltiples plataformas (web, móvil).
- Integración nativa de TypeScript.
- Fácil mantenimiento y testing.

1

INSTALACIÓN Y PRIMEROS PASOS.

Instalación librerias Visual Studio Code.

Instalar las siguientes librerias en el IDE Visual Studio Code:

- ✓ **Angular Essentials (Version x)**
- ✓ **Vscode simpler Icons with Angular**

Instalación de Node.js

Node.js es una librería y entorno de ejecución de JavaScript que nos permite ejecutar código en el servidor, de manera asíncrona, con una arquitectura orientada a eventos, basado en el motor V8 de Google.

El motor V8 compila JavaScript en código máquina nativo en vez de interpretarlo en el navegador, consiguiendo así una velocidad mucho más alta. Node es de código abierto y puede ejecutarse en Mac OS X, Windows y Linux.

Instalación de Node.js

Instalar la última versión estable de **node.js** a través de su página oficial. https://nodejs.org/es

Una vez instalado node.js, abrimos el terminal **VSC** y averiguamos si está instalado el gestor de paquetes **npm.**

- ✓ **node -v** //ver la versión
- ✓ **npm –v** //ver la versión

Para actualizar el paquete de npm:

- ➢ **npm install npm@latest**

Instalación de Angular mediante Angular CLI

- **Angular es la plataforma perfecta para el desarrollo** profesional de aplicaciones modernas. **El CLI es la herramienta** adecuada para generar y ejecutar aplicaciones Angular. Juntos son imbatibles en cuanto a velocidad en desarrollo y a potencia en ejecución.
- El comúnmente conocido como **AngularCLI** o CLI a secas es la herramienta de línea de comandos estándar:
  - Crear nuevos proyectos (ng new).
  - Generar componentes, servicios y módulos (ng generate).
  - Compilar y ejecutar la aplicación (ng serve).

Instalación de Angular mediante Angular CLI

¿Qué nos ofrece Angular CLI?

- ✓ Instalación y esqueleto básico de Angular.
- ✓ Instalación de TypeScript.
- ✓ Comandos para la generación de código (componentes, servicios, etc)
- ✓ Herramientas de minificación y builds para las webapps,
- ✓ Testing.

Para instalar hacemos:

1. Desinstalar los paquetes anteriores de Angular CLI
   - **npm uninstall -g @angular/cli**

2. Instalación de Angular mediante Angular CLI

2. Vaciar la caché del gestor de paquetes npm.
   - **npm cache clean --force**
3. Instalar la última versión de Angular CLI.
   - **npm install -g @angular/cli@latest**
4. Para ver la versión de Angular CLI.
   - **ng v**

Instalación de Angular mediante Angular CLI

Una vez actualizado/instalado Angular CLI ya se puede generar un nuevo proyecto de AngularD

- **ng new NOMBRE_DEL_PROYECTO_ANGULAR** //genera proyecto con standalone=true
- **ng new NOMBRE_DEL_PROYECTO_ANGULAR –standalone=false** //genera proyecto con el fichero app.module.ts

Durante la creación preguntará si quieres crear el **routing** y el **Stylesheet**

- **cd NOMBRE_DEL_PROYECTO_ANGULAR** //cambiar a la carpeta del proyecto
- **ng serve** //lanza el servidor web
- **ng serve -o** //lanza el servidor web y el navegador

La webapp de Angular está en **http://localhost:4200/**

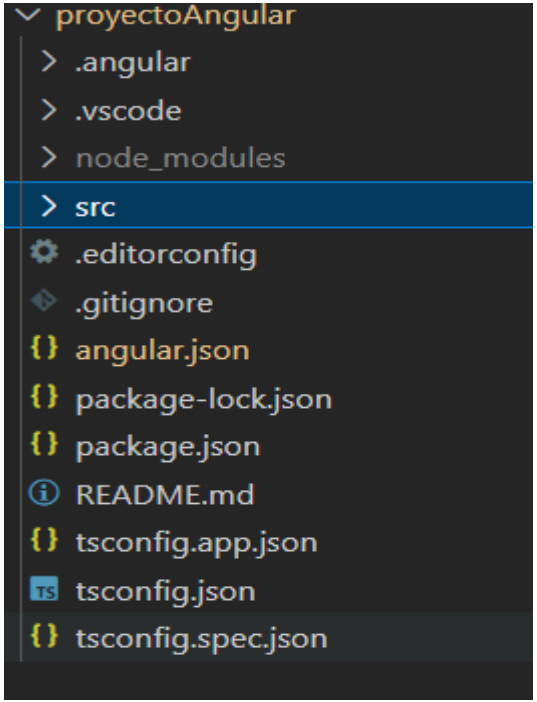
2

ESTRUCTURA Y PARTES DE ANGULAR.

Estructura de Angular

- Un proyecto consta de muchos ficheros: la organización es clave .
- Ficheros generados automáticamente.
- Estructuras de carpetas para organizar.

Estructura de Angular



Ficheros de Angular

Al crear un proyecto en Angular se genera la carpeta **src**:

- **index.html:** Primer fichero en ejecución. Contiene la cabecera con las importaciones e inicia el contenido principal de la aplicación.
- **main.ts**: Es el punto de entrada principal de la aplicación
- **styles.css**: Los estilos principales de la aplicación
- **app/app.component.ts**: Primer componente en ser ejecutado tras index.html.
- **app/app.config.ts**: Es un archivo de configuración de la aplicación.
- **app/app.routes.ts**: En este archivo se definen las rutas de la aplicación.

Ficheros de Angular

- **package.json ->** contiene todas la descripcion de las dependencias y paquetes que npm lee para poder instalarlas en la carpeta ***node_modules.***
- **package-lock.json->** Informa a la aplicación de node como fue creado el fichero package.json
- **tsconfig.json->** se encarga de las configuraciones para TypeScript para la compilación de los archivos .ts.
- **angular.json->** es el archivo que tiene toda la estructura y descripción de nuestro proyecto

Ficheros de Angular

- **editorconfig->** es la configuración del editor y su codificación.
- **gitignore-** es donde se establece que es lo que se va ignorar al momento de hacer git commit y subir cambios al repositorio.
- **node_modules->** contiene todas las dependencias necesario para que nuestro proyecto funcione. Tiene más de 700 paquetes y nos sirve para el desarrollo de la aplicación

Estructura de carpetas

Crearemos dentro de la carpeta **src/app**, una serie de carpetas para almacenar los distintos ficheros que generemos:

- **Interfaces**: Estos elementos nos ayudan en TypeScript a la hora de interactuar con objetos
- **Components**: Los componentes de nuestra aplicación; todo aquello que se pueda usar como tal .
- **Services**:
  - Servicios web
  - Lógica de negocio
  - Interconexión de componentes
- **Otros …**

3

COMPONENTES Y PLANTILLAS.

¿Qué es un componente?

Un componente controla un trozo de pantalla o de la vista. Todo lo que se puede ver en pantalla es controlado y gestionado por este tipo de elementos.

La lógica de un componente dentro de una clase en Angular es que da soporte a una vista, interactuando con ella a través de una API con propiedades y métodos.

El componente hace de mediador entre la vista a través de la plantilla y la lógica de la app donde se incluirá el modelo de datos, es decir una especie de controlador.

Para crear un componente se puede hacer de forma manual o automática.

Forma automática en el terminal sería:

ng generate component nombreComponente

¿Qué es un componente?

o de forma simplificada:

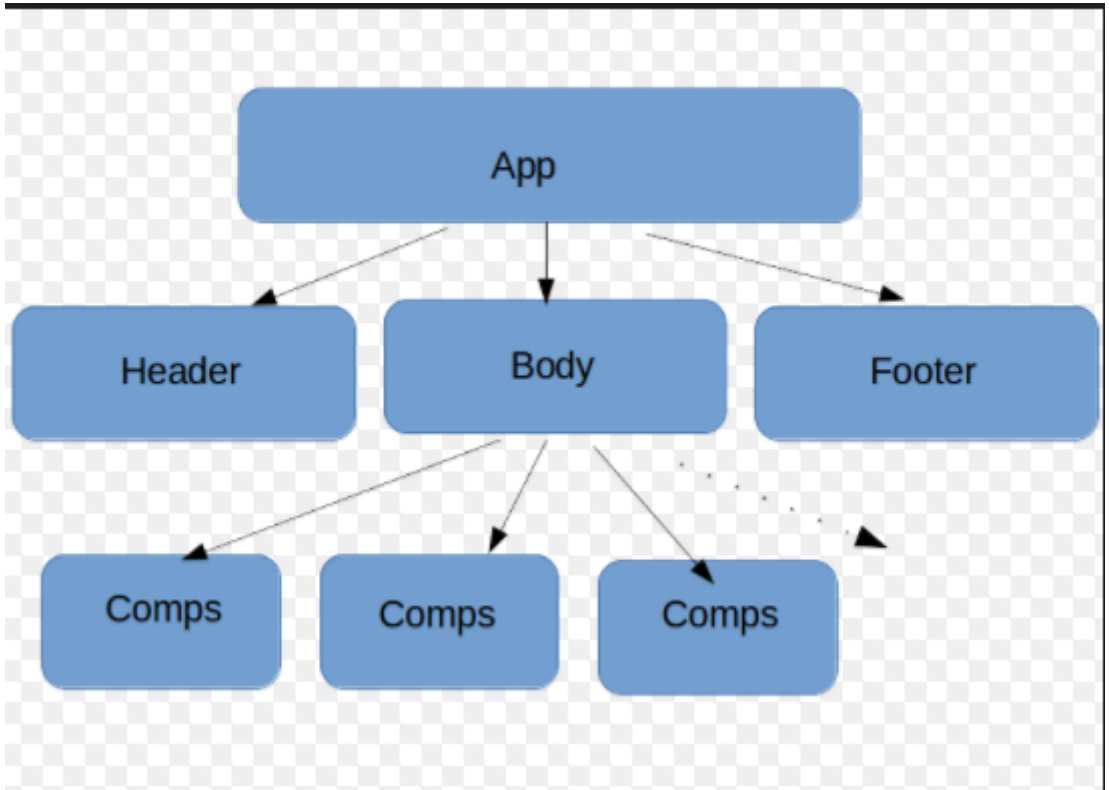
ng g c *nombreComponente*

Se pueden utilizar opciones al crear un componente. Por ejemplo*:*

ng g c nombreComponente **-s**

Esta instrucción crea un componente en la misma ruta donde se escribe la orden sin fichero css.

¿Qué es un componente?



¿Cómo funciona un componente?

Por defecto se encuentra el fichero **app.component.ts**. Es el componente principal de la aplicación. Es el primer componente que se carga y da soporte a otros componentes.

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

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

Importa el módulo **Component**.

Los decoradores aportan características extras a las clases y condicionan el comportamiento.

La clase será el controlador de la plantilla.

¿Cómo funciona un componente?

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

Standalone: comp independiente.

templateUrl: Se asigna una vista o una plantilla.

selector: indica la etiqueta que tiene que cargar el componente.

Import: importa los módulos

styleUrl: Indica los estilos que se van a aplicar a la clase.

EJEMPLOS

Creación de componentes.

4

TYPESCRIPT DENTRO DE ANGULAR.

¿Qué es TypeScript?

- Es un lenguaje de programación libre y de código abierto desarrollado por Microsoft.
- Es un superconjunto de JavaScript, que esencialmente añade tipado estático y objetos basados en clases.
- Extiende la sintaxis de JavaScript, por tanto cualquier código JavaScript existente debería funcionar sin problemas
- Está pensado para grandes proyectos, los cuales a través de un compilador de TypeScript se traduce a código JavaScript original. (se transpila a Js).
- Permite además trabajar sin problemas con famosas librerías de JavaScript como jQuery, MongoDB, Node.js
- Extensiones de los archivos .ts.

Diferencias clave TypeScript vs JavaScript

1. Tipado de variables

javascript

```javascript
// JavaScript permite cambios de tipo
let numero = 42;
numero = "ahora soy texto";  // ✓ Permitido
numero = true;               // ✓ Permitido
numero = { objeto: true };   // ✓ Permitido
```

typescript

```typescript
// TypeScript previene cambios de tipo
let numero: number = 42;
numero = "ahora soy texto";  // ✗ Error de compilación
numero = true;               // ✗ Error de compilación

// Correcto en TypeScript
let numero: number = 42;
let texto: string = "Angular";
let activo: boolean = true;
```

Diferencias clave TypeScript vs JavaScript

2. Tipos básicos en TypeScript

```typescript
// Number
let edad: number = 30;
let precio: number = 19.99;
let hex: number = 0xf00d;

// String
let nombre: string = "María";
let saludo: string = `Hola, ${nombre}`;

// Boolean
let esValido: boolean = true;
let estaActivo: boolean = false;

// Array - Dos formas
let numeros: number[] = [1, 2, 3, 4, 5];
let nombres: Array<string> = ["Ana", "Pedro", "Luis"];

// Tuple (array con tipos específicos)
let persona: [string, number] = ["Carlos", 25];
```

```typescript
// Any (evitar cuando sea posible)
let dato: any = 42;
dato = "texto";  // ✓ Permitido
dato = true;     // ✓ Permitido

// Unknown (más seguro que any)
let valor: unknown = 42;
if (typeof valor === "number") {
  let doble = valor * 2;  // ✓ Permitido tras verificación
}

// Void (para funciones sin retorno)
function saludar(): void {
  console.log("Hola!");
}

// Never (para funciones que nunca retornan)
function error(mensaje: string): never {
  throw new Error(mensaje);
}

// Null y Undefined
let nulo: null = null;
let indefinido: undefined = undefined;
```

Diferencias clave TypeScript vs JavaScript

3. Enums (no existen en JavaScript puro)

javascript

```javascript
// JavaScript: usar objetos congelados
const Colores = Object.freeze({
  ROJO: 0,
  VERDE: 1,
  AZUL: 2
});

let colorFavorito = Colores.ROJO;
```

typescript

```typescript
// TypeScript: Enums nativos
enum Color {
  Rojo,      // 0
  Verde,     // 1
  Azul       // 2
}

let colorFavorito: Color = Color.Rojo;

// Enum con valores personalizados
enum EstadoPedido {
  Pendiente = "PENDIENTE",
  EnProceso = "EN_PROCESO",
  Completado = "COMPLETADO",
  Cancelado = "CANCELADO"
}

let estado: EstadoPedido = EstadoPedido.Pendiente;
```

Diferencias clave TypeScript vs JavaScript

4. Interfaces (no existen en JavaScript)

javascript

```javascript
// JavaScript: sin comprobación de estructura
let usuario = {
  nombre: "Ana",
  edad: 28
};

// Nada previene esto:
usuario.edad = "veintiocho";  // ✓ Error lógico no detectado
```

```typescript
// TypeScript: Interfaces definen contratos
interface Usuario {
  nombre: string;
  edad: number;
  email?: string;  // Propiedad opcional
  readonly id: number;  // Solo lectura
}

let usuario: Usuario = {
  id: 1,
  nombre: "Ana",
  edad: 28
};

usuario.edad = "veintiocho";  // × Error de compilación
usuario.id = 2;  // × Error: readonly
```

Diferencias clave TypeScript vs JavaScript

5. Type Aliases

typescript

```typescript
// Crear tipos personalizados
type ID = string | number;
type Estado = "activo" | "inactivo" | "pendiente";

let userId: ID = "abc123";
userId = 456;  // ✓ Válido

let estadoUsuario: Estado = "activo";
estadoUsuario = "eliminado";  // ✗ Error

// Type para objetos
type Producto = {
  id: number;
  nombre: string;
  precio: number;
  disponible: boolean;
};
```

```typescript
let producto: Producto = {
  id: 1,
  nombre: "Laptop",
  precio: 999.99,
  disponible: true
};
```

Diferencias clave TypeScript vs JavaScript

6. Funciones tipadas

javascript

```javascript
// JavaScript: sin tipos en parámetros ni retorno
function sumar(a, b) {
  return a + b;
}

sumar(5, 3);        // 8
sumar("5", "3");    // "53" - concatenación no esperada
sumar(5);           // NaN - parámetro faltante
```

typescript

```typescript
// TypeScript: tipos en parámetros y retorno
function sumar(a: number, b: number): number {
  return a + b;
}

sumar(5, 3);        // ✓ 8
sumar("5", "3");    // ✗ Error de compilación
sumar(5);           // ✗ Error: falta parámetro

// Parámetros opcionales
function saludar(nombre: string, apellido?: string): string {
  return apellido ? `Hola ${nombre} ${apellido}` : `Hola ${nombre}`;
}
```

Diferencias clave TypeScript vs JavaScript

6. Clases

javascript

```javascript
// JavaScript moderno con clases
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  presentarse() {
    return `Soy ${this.nombre}`;
  }
}
```

typescript

```typescript
// TypeScript: propiedades tipadas y modificadores de acceso
class Persona {
  // Propiedades tipadas
  private nombre: string;
  protected edad: number;
  public email: string;
  readonly dni: string;

  // Constructor tipado
  constructor(nombre: string, edad: number, dni: string) {
    this.nombre = nombre;
    this.edad = edad;
    this.dni = dni;
    this.email = "";
  }

  // Métodos tipados
  public presentarse(): string {
    return `Soy ${this.nombre}`;
  }

  private validarEdad(): boolean {
    return this.edad >= 18;
  }
}
```

5

DIRECTIVAS, BINDING Y EVENTOS

Directivas.

Una directiva son pequeñas funcionalidades que se utilizan en la vista, fichero html, bien para mostrar algo según unas condiciones o para añadir atributos.

Hay directivas estructurales y directivas de atributo.

- **Directivas estructurales**: Se diferencian fácilmente al ser precedidas por un asterisco, alteran el layout: añadiendo, eliminando o reemplazando elementos en el DOM.

- **Directivas de atributo**: Las directivas atributo alteran la apariencia o el comportamiento de un elemento existente en el DOM y su apariencia es igual a la de atributos HTML.

Directivas estructurales.

Directiva nglf, esta directiva es para realizar una condición simple. Ejemplo//

En el fichero .ts

```
const color: boolean = true;
```

En la vista .html

```
<p *nglf='color==true'>tengo color</p>
```

color, es el nombre de variable que se estable en el archivo TS. Con la directiva va a leer automáticamente las variables, si **nglf** envía true va a imprimir la línea, en caso que el resultado sea false, ignorará la línea.

Directivas estructurales.

Como cualquier if se puede decidir con un else de la siguiente manera:

```
<div *ngIf="condición; then thenBlock else elseBlock"></div>

<ng-template #thenBlock>...</ng-template>
<ng-template #elseBlock>...</ng-template>
```

Directivas estructurales.

En la versión 17 se ha incorporado el @if para admitir bloques de flujo de control:

```
@if (a > b) {
  {{a}} is greater than {{b}}
}
```

También se permite que @if pueda tener una o mas ramas asociadas

```
@if (a > b) {
  {{a}} is greater than {{b}}
} @else if (b > a) {
  {{a}} is less than {{b}}
} @else {
  {{a}} is equal to {{b}}
}
```

Directivas estructurales.

Directiva ngFor, se usa para hacer un bucle de elementos. Por ejemplo, hacer una **lista dinámica, o un bloque de información.**

La sintaxis es:

*ngFor="let <value> of <collection>" .

<value> es un nombre de variable de su elección. <collection> es una propiedad en su componente que contiene una colección, generalmente una matriz, pero puede ser cualquier cosa que pueda repetirse en un bucle.

A veces, también se desea obtener el *índice* del elemento en la matriz sobre la que se está iterando, entonces se realizará de la siguiente forma:

*ngFor="let <value> of <collection; let i=index>"

Directivas estructurales.

En la versión 17 se ha incorporado el @for :

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

Directivas estructurales.

Directiva ngSwitch, es similar a nglf y es como el switch de la programación. Es decir, que permite que entre varios conjuntos de tags solo esté uno de ellos, borrando los que no cumplen la condición.

La sintaxis es:

<div [ngSwitch]="variable"> .

Para trabajar con la directiva ngSwitch también hay que utilizar la directiva ***ngSwitchCase,** que son los casos donde concuerde la evaluación.

La sintaxis es:

*<div *ngSwitchCase="0">*

Directivas estructurales.

En la versión 17 se ha incorporado el @switch :

```
@switch (condition) {
  @case (caseA) {
    Case A.
  }
  @case (caseB) {
    Case B.
  }
  @default {
    Default case.
  }
}
```

Directivas atributo.

Directiva ngSytle, permite establecer propiedades de estilo de elementos DOM.

La sintaxis es:

< **etiqueta** [ngStyle]="{' background-color ' : ' green '}" > < / < **etiqueta** >

Directiva ngClass, permite añadir/eliminar varias clases a un elemento de forma simultánea y dinámica.

.

La sintaxis es:

< **etiqueta** [ngClass]="{'clase1' : condicion, 'clase2':condicion . . .}" > < / < **etiqueta** >

Eventos.

Para establecer eventos en Angular se realiza de la siguiente forma:

Sintaxis:

<etiqueta (**evento**)=“function()">

Código en la vista .hmtl:

<select id="listCenTra“ (change)=“mostrar()"

Código en el componente .ts:

```
mostrar() {
    // Aquí irá la lógica
}
```

Doble binding. ngModel

El *doble binding* o enlace vista controlador en ambos sentidos es una de las claves del éxito de Angular. Para trabajar esta técnica hay que usar la directiva **ngModel**. Esta enlaza **una propiedad del modelo con un control de la vista,** un input, textarea, etc. De manera *automágica* Angular se suscribe a cambios en el DOM y observa el estado del modelo manteniéndolos en sincronía.

Para que funcione binding hay que importar en el componente el **FormsModule**, en el fichero **app.module.ts**.

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

Two Data binding. ngModel

También hay que importarlo en el decorador @NgModule

```
@NgModule({
  declarations: [
    AppComponent,
    FrutasComponent,
    EmpleadosComponent,
    EmpleadosListComponent

  ],
  imports: [
    BrowserModule,
    FormsModule

  ],
```

Sintaxis:

<etiqueta **[(ngModel)]="**propiedad“></etiqueta>

Ejemplo//

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

Hola {{nombreDelProgramador}}

6

PIPES.

Pipes.

Las Pipes o filtros son pequeñas funciones que se utilizan en las vistas, por ejemplo, si deseamos mostrar el formato de una moneda, redondear a la cantidad de decimales que requerimos, convertir una cadena a mayúsculas o minúsculas, etc. También podemos crear Pipes personalizados.

Pipes.

DatePipe

Nos permite mostrar las fechas de una forma mas elegante y agradable a la vista. Este Pipe tiene muchos parámetros de configuración pero nos vamos a basar en estructuras sencillas.

```
<div>
  Fecha larga {{fecha}}
  <br>
  Fecha II: {{fecha | date: "longDate"}}
  <br>
  Fecha II: {{fecha | date: "shortDate"}}
  <br>
  Tiempo: {{fecha | date: "hh:mm:ss"}}
</div>
```

Pipes.

- **upperCasePipe**

Este pipe convierte la variable de tipo texto a **mayúscula**.

- **lowerCasePipe**

Este pipe convierte la variable de tipo texto a **minúscula**.

- **titleCasePipe**

Este pipe capitaliza las primeras letras de nuestra variable de tipo texto a **mayúscula**.

```
<div>
  mayúsculas: {{nombre | uppercase}}
  <br>
  minusculas: {{nombre | lowercase}}
  <br>
  Title: {{nombre | titlecase}}
</div>
```

Pipes.

- **slicePipe**

El pipe Slice podemos extraer elementos según indiquemos los parámetros del pipe. Podemos cortar desde Arrays, strings o listas.

Sintaxis:

{{ variable | slice:posicionInicial:posicionFinal }}

```
<div>
  Extrae los dos primeros caracteres: {{nombre | slice:1:3 }}
</div>
```

Pipes.

DecimalPipe

Permite mostrar números en un formato especifico. Es un poco incongruente que el pipe se llame decimal y lo utilicemos con el nombre number.

Sintaxis:

{{ var | number:' minimoDeEnteros . minimDeDecimales -maximoDeDecimales ' }}

Los parámetros deben estar en formato de string ''.

```
<div>
  <br>Formater un numero: {{numero | number:'3.1-2' }}
</div>
```

Formatea un número con 3 dígitos en la parte entera y 1 o dos decimales

Pipes.

CurrencyPipe

Este pipe transforma números a formato de monedas Locales basado en las normas ISO 4217.

```
<div>
  <br>Formater una cantidad de dinero: {{numero | currency:'EUR':true:'1.1-2' }}
</div>
```

Pipes. Internacionalización

La internacionalización es el proceso de diseño y preparación de la aplicación para que pueda utilizarse en varios idiomas. La localización es el proceso de traducir la aplicación internacionalizada a idiomas específicos.

En el fichero **app.config.ts** hay que realizar las siguientes incorporaciones.

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

// Internacionalización española
import { registerLocaleData } from '@angular/common';
import localeEs from '@angular/common/locales/es';
registerLocaleData(localeEs, 'es');
```

Pipes. Internacionalización

Por último en **providers**, incluir la variable importada:

providers: [{ provide: LOCALE_ID, useValue: 'es' }],

Crear Pipes personalizadas.

Lo primero que haremos será crear un **directorio llamado pipes,** dentro del directorio app del proyecto Angular. A continuación, en este directorio se creará un fichero

Se podrá hacer manualmente o de forma automática:

ng g p nombrePipe

Este fichero llevará la clase de nuestro filtro personalizado.

Crear Pipes personalizadas.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'pipeFiltList',
  standalone: true
})
export class PipeFiltListPipe implements PipeTransform {

  transform(lista: number[], min:number,max: number): number[] {
    if (!lista && !max && !min){
      return lista
    }

    lista= lista.filter(element=>element>=min && element<=max);
    return lista.sort((a,b)=> a-b)
  }

}
```

Importar los componentes de la pipes

El nombre de la Pipe

Definición de la clase implementando la interface

Los parámetros que recibe la pipe

Retornar el valor formateado

Crear Pipes personalizadas.

Seguidamente solo falta usarla en un componente. Para ello se importará en el fichero donde se utilizará, **app.xxxxx.ts**

```
imports: [CommonModule,PipeFiltListPipe],
```

En la vista se indicará la pipe de la siguiente forma:

```
<tr>
  <th>14</th>
  <td>{{lista}}</td>
  <td>Pipe Filtra</td>
  <td>{{lista | pipeFiltList:4:33}}</td>
</tr>
```

7

ROUTING Y NAVEGACIÓN.

Routing.

Angular permite una navegación muy sencilla. Con Angular **routing** se puede dividir la aplicación en diferentes áreas que podemos llamar páginas.

Para ello se utiliza el fichero **app.routes.ts** que se genera al crear un proyecto.

Generalmente se utiliza para definir las rutas de la aplicación. Este archivo es parte del enrutamiento que Angular proporciona para crear aplicaciones de una sola página (SPA)

Routing.

Para trabajar con rutas podemos hacerlo de dos formas:

1. **Enrutamiento nomal:**
    - Todos los módulos se cargan inicialmente al inicio de la aplicación, independientemente de si son o no necesarios.
    - Esto puedo resultar en tiempos de carga más largos y un mayor uso de recursos al cargar toda la aplicación.

Routing.

2. **Lazy Loading:**
    - Solo los módulos necesarios para la vista actual se cargan dinámicamente cuando se necesitan.
    - Mejora el tiempo de carga inicial y reduce el uso de recursos al cargar solo lo esencial al inicio.

Routing. app-routes.ts (Enrutamiento normal)

```
import { Routes, RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';

import { NgSwitchComponent } from './components/ng-switch/ng-switch.component';
import { EmpleadoComponent } from './components/empleado/empleado.component';
import { NgClassComponent } from './components/ng-class/ng-class.component';
import { NgStyleComponent } from './components/ng-style/ng-style.component';
import { NgIfForComponent } from './components/ng-if-for/ng-if-for.component';
import { DataBindingComponent } from './components/data-binding/data-binding.co
import { BodyComponent } from './components/body/body.component';
const routes: Routes = [
  { path: '', component: BodyComponent },
  { path: 'body', component: BodyComponent },
  { path: 'ngswitch', component: NgSwitchComponent },
  { path: 'empleado', component: EmpleadoComponent },
  { path: 'ngclass', component: NgClassComponent },
  { path: 'ngstyle', component: NgStyleComponent },
  { path: 'ngiffor', component: NgIfForComponent },
  { path: 'databind', component: DataBindingComponent },
  { path: '**', component: BodyComponent }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Importamos los módulos que necesitamos para configurar las rutas.

Importamos los componentes que vamos a enrutar.

Un array de objetos json con la configuración de la ruta.

Importamos el array routes

Routing. app-routes.ts (Lazy loading)

```
import { Routes } from '@angular/router';

export const routes: Routes = [
  {
    path: 'body',
    loadComponent: () => import('./components/body/body.component').then(c=>c.BodyComponent)
  },
  {
    path: 'ngiffor',
     loadComponent: () => import('./components/ng-if-for16/ng-if-for16.component').then(c=>c.NgIfFor16Component)
  },
```

Routing. app.component.ts

```
<app-header />
<div class="container">
    <router-outlet />
</div>
<app-footer />
```

En este componente se cargan dinámicamente los componentes asociados a las rutas definidas en la aplicación

Routing. app.xxxxxxx.html

```
<div class="dropdown-menu" aria-labelledby="navbarDropdown">
  <!-- Opciones en el menú desplegable -->
  <a routerLink="/ngiffor" routerLinkActive="active" class="d
  <a routerLink="/ngswitch" routerLinkActive="active" class="dr pdown-item" href="#">>ngSwitch</a>
  <div class="dropdown-divider"></div>
  <a routerLink= 'iffor17'routerLinkActive="active" class="dropdown-item" href="#">if for</a>
  <a routerLink= 'switch17'routerLinkActive="active" class="dropdown-item" href="#">switch</a>
</div>
```

Menú de navegación. Se especifica **routerli** donde se indica la ruta

8

FORMULARIOS REACTIVOS.

Formularios.

1. Importar el módulo **ReactiveFormsModule** de **@angular/forms** y añadir en **imports**. :

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

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

@Component({
  selector: 'app-login',
  imports: [ ReactiveFormsModule],
  templateUrl: './login.html',
  styleUrl: './login.css',
})
export class Login {
```

Formularios.

```
import { Component, OnInit } from "@angular/core";
import { FormGroup, FormBuilder, Validators } from "@angular/forms";

@Component({
  selector: "app-formulario",
  templateUrl: "./formulario.component.html",
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent implements OnInit {
  frmDatos: FormGroup;

  constructor(private frmBuilder: FormBuilder) {
    this.frmDatos = frmBuilder.group({
      nomApe: [
        "",
        [Validators.required, Validators.pattern(/^[a-zá-úªñ\s]+$/i)]
      ],
      email: ["", [Validators.required, Validators.email]],
      passw: [
        "",
        [Validators.required, Validators.minLength(8), Validators.maxLength(25)]
      ]
    });

  }
```

Importar las API para trabajar con los formularios reactivos

Crear un objeto de tipo FormGroup que hace referencia al formulario de la vista.

Establecer las reglas de validación

Formularios.

Angular, provee la clase **Validators** que trae funciones comunes que son de mucha utilidad en los formularios:

Validators.required = Comprueba que el campo sea llenado.

Validators.minLength = Comprueba que el campo cumpla con un mínimo de caracteres.

Validators.maxLength = Comprueba que el campo cumpla con un máximo de caracteres.

Validators.pattern = Comprueba que el campo cumpla con un patrón usando una expresión regular.

Validators.email = Comprueba que el campo cumpla con un patrón de correo válido.

Con estas funciones podemos crear conjuntos de validaciones

Formularios.

Con estas funciones podemos crear conjuntos de validaciones:

'company': ['', [Validators.required, Validators.minLength(5), Validators.maxLength(10)]],

Formularios.

A continuación, en la vista hay que controlar cómo mostrar los errores al usuario a partir de el formulario '**formulario**' declarada en el controlador.

```
<mat-card-content>
  <form [formGroup]="loginFrm" (ngSubmit)="onLogin()">

    <mat-form-field appearance="outline" class="full-width">
      <mat-label>Email</mat-label>
      <mat-icon matPrefix>mail</mat-icon>
      <input matInput formControlName="email" type="email" autocomplete="username">
      @if (email.errors?.['required'] && email.touched) {
        <mat-error>El email es requerido</mat-error>
      }
      @if (email.errors?.['email']) {
        <mat-error>Ingrese un email válido</mat-error>
      }
    </mat-form-field>
```

En formGroup se indica el nombre establecido en el componente

formControlName es el nombre del campo donde se realizan las validaciones .

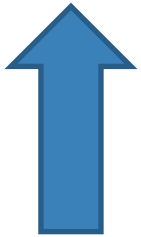
Formularios.

Como mostrar mensajes de error según las validaciones indicadas en el fichero .ts:

```
@if (email.errors?.['required'] && email.touched) {
  <mat-error>El email es requerido</mat-error>
}
@if (email.errors?.['email']) {
  <mat-error>Ingrese un email válido</mat-error>
}
```

Formularios.

```
<button type="submit" [disabled]="frmDatos.invalid" class="btn btn-primary">Enviar</button>
```



El botón está desactivado mientras el formulario tenga datos erróneos.

Formularios.

Para tratar la información una vez que se envía el formulario se realizará de la siguiente forma:

```
<form [formGroup]="frmDatos" (submit)="envioDatos(frmDatos)">
```

El evento submit ejecutará una función **envioDatos** dónde se pasará los datos del formulario. La función se desarrollará en el fichero .ts

9

El constructor. Modelos de datos, Objetos e Interfaces

Modelo de datos.

- La manera más común de definir tipos, para las personas acostumbradas a lenguajes de programación orientados a objetos, son las clases.
- En angular ese modelo de datos se puede representar mediante una clase una clase o un interface.
  - ✓ Crear una clase: **ng g class <nombre>**

```
export class CEmpleados {
  constructor(
    public nombre: string,
    public edad: number,
    public cargo: string,
    public contratado: boolean
  ) {}
}
```

Modelo de datos.

- Crear un interfaz: **ng g i <nombre>**

```
export interface Empleados {
    nombre: string,
    edad: number,
    cargo: string,
    contratado: boolean
}
```

Modelo de datos.

- Si ambas construcciones sirven para más o menos lo mismo ¿cuándo usar clases y cuándo usar interfaces? la respuesta depende un poco sobre cómo vayamos a generar los datos.
- Es muy habitual usar simplemente interfaces, desprovistas de inicialización y funcionalidad, ya que esas partes es común que sean delegadas en los servicios. Pero en el caso de usar clases, los nuevos objetos serán creados con la palabra "new".
- Las diferencias es que la interface solo está en tiempo de compilación. Esto solo le permite verificar que los datos esperados, recibidos sigan una estructura particular. En cambio la clase está presente en el tiempo de ejecución y puede definir más métodos en ellas con el procesamiento si fuese necesario.

Crear clases, modelos de datos y objetos.

2. Importar el modelo de datos en el componente.

```
import { Component, OnInit } from "@angular/core";
import { CEmpleado, Empleado } from "../../modelos/empleados";

@Component({
  selector: "app-empleados",
  templateUrl: "./empleados.component.html",
  styles: []
})
export class EmpleadosComponent implements OnInit {
  aListEmp: Array<CEmpleado>;
  aListaEmpInterf: Array<Empleado>;
  constructor() {
    this.aListEmp = [
      new CEmpleado("Nuria", 23, "testing", true),
      new CEmpleado("Javier", 33, "Analista", false),
      new CEmpleado("Manuel", 20, "programador", true)
    ];
    this.aListaEmpInterf = [
      { nombre: "Nuria", edad: 23, cargo: "testing", contratado: true },
      { nombre: "Javier", edad: 33, cargo: "analista", contratado: false },
      { nombre: "Manuel", edad: 20, cargo: "programador", contratado: true }
    ];
  }
  ngOnInit() {}
}
```

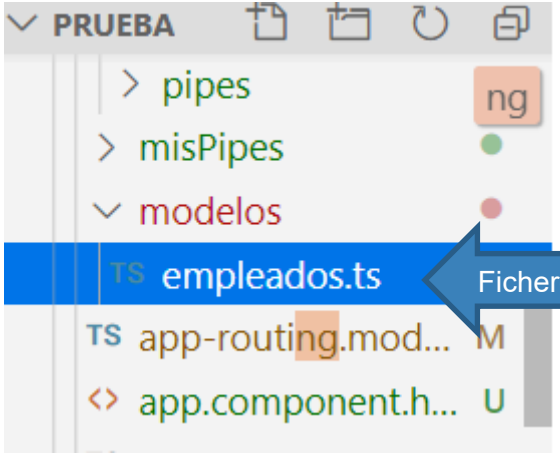
Importar los modelos de datos.

Se crean 2 arrays. El primero con una clase y el segundo con un interface.

Asignación de datos.

Crear clases, modelos de datos y objetos.

1. Vamos a crear un modelo para el componente **empleados,** que será una estructura básica de las propiedades que va a tener ese objeto y ese interface.




```
export interface Empleado {
    nombre: string,
    edad: number,
    cargo: string,
    contratado: boolean
}
export class CEmpleado {
    // public nombre:string;
    // public edad:number;
    // public cargo:string;
    // public contratado:boolean;
    // constructor(nom, edad, cargo, contr){
    //     this.nombre= nom,
    //     this.edad= edad,
    //     this.cargo= cargo,
    //     this.contratado= contr
    // }
    constructor(
        public nombre:string,
        public edad:number,
        public cargo:string,
        public contratado:boolean
    ){}

}
```

Crear una interface

Crear una clase.Forma clásica. Establecer propiedades y asignar valores

Angular permite simplificar

10

COMUNICACIÓN ENTRE COMPONENTES @Input, @Output.

Comunicación entre componentes.

En Angular hay varias formas de facilitar a los componentes que compartan datos. En este punto vamos a ver la comunicación que puede surgir de los padres a los hijos y viceversa.

Comunicación entre padre a hijo.

En la llamada al componente hijo se especifican la/s propiedades que se van a enviar por medio de bindign, usando corchetes, [**propiedad]=“expresión”[propiedad2]= =“expresión2”]….** A continuación, se establece la variable que se desea enviar, s**endPadre**).

Componente padre (vista)

<app-hijo **[paramPadre]="sendPadre**“></app-hijo>

Comunicación entre padre a hijo.

Seguidamente hay que declarar @input en el componente hijo. Este decorador servirá como el nexo de unión entre los dos componentes. Este decorador se asigna a una variable, la cual podrá ser utilizada en el html del componente hijo.

Componente hijo (componente .ts)

```
@Input() paramPadre?:string
```

El decorador **paramPadre** guardará el dato pasados desde el padre. Para utilizarlo en el html tan solo es necesario llamarlo {()}

Comunicación entre hijo a padre.

Si el componente hijo desea mandarle cualquier información al padre, se podrá realizar mediante el decorador @Output que es de tipo EventEmitter()

Componente hijo (componente .ts)

```
@Output() paramHijo=new EventEmitter<string>()

...

  public sincronPadre(){

    this.paramHijo.emit(sendHijo)

  }
```

Comunicación entre hijo a padre.

Desde el componente padre dentro de la etiqueta asociada al elemento hijo se declara un evento con el mimo nombre del decorador @Outpu declarado en el elemento hijo, y se iguala a la variable que tiene que recibir como parámetro un evento ($event) que es aquello emitido por EventEmitter del hijo.

```
<app-hijo [paramPadre]="sendPadre"
    (paramHijo)="nombre=$event"></app-hijo>
```

11

SERVICIOS.

Servicios.

Los servicios en Angular sirven para implementar la parte de recogida y envío de datos, es decir, se recomienda que los componentes no traten con la comunicación con servidores o datos directamente. Lo es mejor es delegar esta funcionalidad a los servicios.

Servicios.

La particularidad de las clases de servicios está en su decorador: @Injectable(). Esta función viene en el @angular/core e **indica que esta clase puede ser inyectada** dinámicamente a quien la demande. Aunque es muy sencillo crearlos a mano, el CLI nos ofrece su comando especializado para crear servicios:

ng g s services/servicios //crea el fichero **servicios.service.ts** en la carpeta services

Servicios. xxxx.service.ts

```
import { Injectable } from '@angular/core';
import { User } from '../modulos/users';

@Injectable({
  providedIn: 'root'
})
export class UsersService {
  public aUsers:Array<User>
  constructor() {
    this.aUsers=[
      {
        "id": 1,
        "email": "george.bluth@reqres.in",
        "first_name": "George",
        "last_name": "Bluth",
        "avatar": "https://reqres.in/img/faces/1-image.jpg"
      }
    ]
  }
}
```

Se importa la función Inyectable que se utiliza para decorar la clase del servicio y el modelo User

Este decorador se aplica a la clase UsersService y le indica que estará disponible para toda la aplicación

Servicios. fichero.component.ts –forma tradicional-

```ts
import { Component, inject } from '@angular/core';
import { UsersService } from '../../services/users.service';
import {User} from '../../modulos/users'


@Component({
  selector: 'app-servicios',
  standalone: true,
  imports: [],
  templateUrl: './servicios.component.html',
  styles: ``
})
export class ServiciosComponent {
  public aUsuarios!:Array<User>; //Declarar
  constructor(private serv_usuarios:UsersService){

    this.aUsuarios=this.serv_usuarios.aUsers;
  }
```

Se importa el fichero que tiene el servicio

Se crea un servicio

Se crea una igualdad con la lista del servicio

Servicios. fichero.component.ts –forma actual v.17-

```
import { Component, inject } from '@angular/core';
import { UsersService } from '../../services/users.service';


@Component({
  selector: 'app-servicios',
  standalone: true,
  imports: [],
  templateUrl: './servicios.component.html',
  styles: ``
})
export class ServiciosComponent {
  public aUsuarios= inject(UsersService)
}
```

Se importa el fichero que tiene el servicio

Se inyecta el servicio

12

API REST y HTTP .

API REST.

Una de las tareas más comunes cuando se trabaja Angular es consumir servicios para poder obtener datos, y eso supone en la mayoría de los casos tener un servicio al cual conectar para realizar las típicas operaciones REST.

Un API REST es una interfaz donde podemos conectar nuestras aplicaciones para obtener, crear, actualizar o eliminar datos de forma sencilla, ya sea como clientes, o como proveedores del servicio. El API REST permitir a otros realizar estas acciones gracias a los distintos métodos básicos (o principales) de HTTP (POST, GET, PUT, DELETE).

API REST.

Las acciones a realizar son las siguientes:

- GET sirve para obtener
- POST para crear
- PUT para actualizar
- DELETE para eliminar.

Servicios. Http con AJAX.

Las comunicaciones *http* son una pieza fundamental del desarrollo web, y en **Angular** siempre han sido fáciles y potentes, sobre todo en la última versión de angular.

Para poder trabajar con http hay que trabajar con los *observables* y los servicios de la librería **@angular/common/http** con los que realizar **comunicaciones asíncronas en Angular**.

Al final podremos almacenar y recuperar los datos consumiendo un servicio REST.

Servicio HttpClient

La librería **@angular/common/http** trae el módulo **provideHttpClient** con el servicio inyectable **HttpClient** que hay que declarar como dependencia de los constructores.

El servicio **HttpClient** nos va a permitir realizar solicitudes HTTP (GET, POST, PUT, DELETE) de manera sencilla y eficiente.

Está diseñado para trabajar con observables de RxJS, lo que facilita la gestión de respuestas asíncronas y la manipulación de datos.

Servicio HttpClient

Lo primero que debemos hacer es importar este módulo **provideHttpClient** en el fichero **app.config.ts**

```
import {provideHttpClient, withFetch} from '@angular/common/http';
```

Añadir en el método **appConfig**: **provideHttpClient**(withFetch())

A continuación, ya podemos usar el **HttpClient** en cualquier componente de nuestra aplicación, ahora vamos a usarlo en un servicio de Angular

Servicio HttpClient

Crear un servicio, **xxxxx.service.ts**, en la carpeta **services.**

En el servicio hay que importar **HttpClient**(para hacer las peticiones al API REST por AJAX) y **HttpHeaders**(para establecer cabeceras en las peticiones, **HttpErrorResponse** (permite una respuesta a un error producido por un observable.):

```
import {HttpClient, HttpHeaders, HttpErrorResponse} from '@angular/common/http';
```

Servicio HttpClient

La librería **RxJS** proporciona una implementación del tipo Observable, que se necesita hasta que el tipo se convierta en parte del idioma y hasta que los navegadores lo admitan.

Lo siguiente es importar el **Observable**, este es un método muy importante, ya que para decirlo de manera muy simple, **observable no bloqueara el hilo principal**, simplemente estará pendiente cuando regrese algún dato que nos sirva.

- import { Observable, of } from "rxjs";

También importamos operadores para manipular las colecciones de datos de los observables.

- import { map, catchError, tap } from "rxjs/operators";

Servicio HttpClient

Fichero de servicios I:

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

import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Observable, of } from "rxjs";
import { catchError } from "rxjs/operators";

import { User } from "../modelos/usuario";

// cabeceras
const cabecera = {
  headers: new HttpHeaders({ "Content-Type": "application/json" })
};
```

Importar los módulos necesarios para realizar la conexión http.

Importar el observable, que permite trabajar con los datos de forma más compleja. También se importa operadores para manejar errores.

Cabecera para los métodos: post, put y delete

Servicio HttpClient

Fichero de servicios II:

```
export class UsersService {
  private usersUrl: string; // establecer la ruta
  constructor(private http: HttpClient) {
    this.usersUrl = "/api/users";
  }
  // extraer todos los usuarios
  getUsers(): Observable<User[]> {
    return this.http.get<User[]>(this.usersUrl);
  }
  // extraer el usuario del id pasado como parámetro
  getUser(id): Observable<User> {
    return this.http.get(this.usersUrl + '/' + id).pipe(
      catchError(this.handleError<any>("getUser"))
    );
  }
```

Servicio HttpClient

Fichero de servicios III:

```
// borrar el usuario del id pasado como parámetro
deleteUser (id: number): Observable<{}> {
  const url = `${this.usersUrl}/${id}`; // Elimina api/user/?
  return this.http.delete(url, cabecera)
    .pipe(
      catchError(this.handleError('deleteUser'))
    );
}
// añadir el usuario  pasado como parámetro
addUser (user: User): Observable<User> {
  return this.http.post<User>(this.usersUrl, user, cabecera)
    .pipe(
      catchError(this.handleError('addUser', user))
    );
}
```

Servicio HttpClient

Fichero de servicios IV:

```
updateUser (user: User): Observable<User> {
  return this.http.put<User>(this.usersUrl, user, cabecera)
    .pipe(
      catchError(this.handleError('updateUser', user))
    );
}

// manejador de errores
private handleError<T>(operation = "operation", result?: T) {
  return (error: any): Observable<T> => {
    // envíar el error de forma instantanea
    console.error(error); // enviar a la consola
    // TODO: Mandar más información sobre el mensaje producido
    console.log(`${operation} failed: ${error.message}`);
    // Deja que la aplicación siga funcionando devolviendo un resultado vacío
    return of(result as T);
  };
}
```

Establecer el manejador de errores.

Fichero componente.component.ts

- Llamar a la función del servicio que recoge todos los usuarios del API REST:

```
this._usrservices.getUsers().subscribe((data: {}) => {
  console.log(data);
  this.usuarios = data; // el objeto de datos devuelto por AP
});
```

Fichero componente.component.ts

- Invocar a la función del servicio que elimina el registro que indica el parámetro:

```
this._usrservices.deleteUser(id).subscribe(
  res => {
    this.getUsuarios();
  },
  err => {
    Swal.fire(err, "", "error");
  }
);
```

Servicio HttpClient

Fichero componente.component.ts

- Invocar a la función del servicio que añade un registro:

```
this._userservice.addUser(this.usuario).subscribe(data => {
  Swal.fire("Usuario Insertado", "", "success").then(value => {
    this.router.navigate(["/manten"]);
  });
},
error => Swal.fire("El usuario no pudo ser insertado", error, "warning")
);
```

Servicio HttpClient

Fichero componente.component.ts

- Invocar a la función del servicio que actualiza un registro:

```
this._userservice.updateUser(this.usuario).subscribe(data => {
  Swal.fire("Usuario Modificado", "", "success").then(value => {
    this.router.navigate(["/manten"]);
  });
},
error => Swal.fire("El usuario no pudo ser modificado", error, "warning")
);
```

13

USAR Bootstrap y FontAwesome, sweetAlert2 con ANGULAR.

Instalación

Hay dos formas de integrar este tipo de librerias con Angular:

1. Agregar las librería al proyecto de Angular de forma local o mediante CDN:

2. Mediante **npm,** instalando y configurando dependencias.

Agregar las librerías al proyecto de Angular

Incluir las librería en el fichero **index.html.** Se puede hacer descargando localmente como se ha realizado con JavaScript “puro” o usando un cdn.

Local:

```
<link rel="stylesheet" href=../bootstrap/css/bootstrap.min.css >
<script src=../bootstrap/css/bootstrap.min.js ></script>
```

CDN:

```
<link rel="stylesheet" href=https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css >
<script src=https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js ></script>
```

Agregar las librerías al proyecto de Angular

1. Instalar bootstrap con npm.

   npm install bootstrap

1. Instalar FontAwesome

   npm install @fortawesome/fontawesome-free

2. Instalar SweetAlert 2

   npm install sweetalert2

Agregar las librerías al proyecto de Angular

5. Abrir el fichero **angular.json** y añadir en la etiqueta **styles** y **script** lo siguiente:

```
"styles": [
  "./node_modules/bootstrap/dist/css/bootstrap.min.css",
  "./node_modules/@fortawesome/fontawesome-free/css/all.min.css",
  "src/styles.css"
],
"scripts": [
  "./node_modules/jquery/dist/jquery.min.js",
  "./node_modules/bootstrap/dist/js/bootstrap.min.js",
  "./node_modules/@fortawesome/fontawesome-free/js/all.min.js"
]
```

14

Despliegue de aplicaciones a producción

Despliegue de aplicaciones a producción

Una vez que el código está 100% revisado y no hay problemas de ejecución, lo primero que a tener en cuenta es tener una aplicación libre de dependencias innecesarias, recordad que cada dependencia que se agrega es código que será conjuntado a la aplicación, es por ello revisar el archivo **package.json** para corroborar que hay librerías innecesarias.

Modificar en el archivo **environments.ts,** la propiedad **production** a true

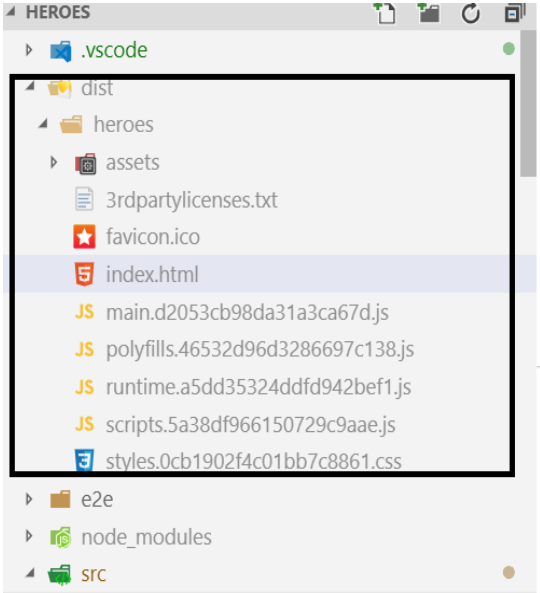
```
export const environment = {
  production: true
};
```

Una vez finalizada esa revisión hay que subirla a nuestro servidor real en Internet, para esto hay que ejecutar el siguiente comando de Angular CLI:

```
ng build --prod
```

Despliegue de aplicaciones a producción

A continuación se genera una carpeta llamada 'dist' que contiene todos los archivos que debemos subir a nuestro servidor de Internet.



Despliegue de aplicaciones a producción

Los archivos de esta carpeta se deben subir a la carpeta raíz del servidor de Internet.

Si la aplicación Angular no se ejecuta en la raíz del servidor, el proceso de compilación es diferente. Por ejemplo, la aplicación se va a cargar en una carpeta que se llama **ejemplos,** la compilación será.

ng build --prod --base-href=/ejemplos/

CRÉDITOS

Mª Luz Sánchez Rubio.

Módulo: Desarrollo Web en Entorno Cliente

Dpto. Informática –IES TRASSIERRA-