Angular es para aplicaciones de una sola página SPA.

Angular 8 node.js es el framework - descargarlo e instalar

Typescript es un lenguaje de alto nivel con clases y otras cosas, que al compilar se traduce en Javascript

Angular CLI : un intérprete de comandos de Angular. Facilita inicio y creación de esqueleto. Para instalar:

npm install -g @angular/cli

npm -v para ver la versión

ng new curso-angular crea un nuevo proyecto

ng serve --open inicia la aplicacion abriendolo en el navegador. Abritá localhost:4200

Ficheros del proyecto

- README.md serviría para dejar documntación sobre el proyecto y debería leerse por cualquiera que comience con la aplicación

- tsconfig.json y tslint.json sirven para configurar Typescript y TLint respectivamente. Por ejemplo, para configurar los errores que puede dar.

- package.json con dependencias en las carpetas node\_modules. Cuando le pasemos el proyecto a otra persona, podrá instalar con una sola instruccion todas las dependencias, gracias a este fichero.

- package-lock.json mejor no tocar, contiene la versión instalada

- e2e ficheros de testing de nuestra aplicacion

- .editorconfig configuracion con varios editores de texto

- .gitignore para decirle a git qué no debe subir, por ejemplo /node\_modules

- angular.json configuracion de Angular CLI para algo más avanzado

- En la carpeta src estarán todos nuestros archivos:

-- Otros ficheros de configuración de tsconfig

-- test.tsconfig

-- index.html - al ser una aplicacion SPA irá todo el proyecto

-- assets todos los recursos estáticos

-- enviroments hay para los diferentes entornos como DES y PRO o diferentes ramas.

-- app/archivos la carpeta con los componentes

--- app.module es por defecto el módulo inicial, pero por defecto Angular modulariza todo y deberíamos separar cuando sea necesario

--- app.component son los 4 ficheros que componen el componente

App.module.ts define el módulo principal AppModule donde se definen los componentes y todo lo que usa.

# Componentes

Básicamente tiene imports, un decorador y luego el código de la clase.

### En el decorador:

El Selector tiene que ser único en toda la aplicación

templateUrl - la página html que vamos a usar

styleUrls - el CSS

Clase y una variable

ng generate componente nombre-componente - sirve para crear más componentes , aunque sólo consiste en crear un nuevo fichero como el antiguo

# Bootstrap

Bootstrap sirve para organizar y diseñar mejor una pagina web. Dentro de nuestro proyecto ya abierto

1. npm install --save bootstrap
2. Añadir en el fichero angular.json: "./node\_modules/bootstrap/dist/css/bootstrap.min.css"
3. npm install --save @ng-bootstrap/ng-bootstrap

Bootstrap define la pantalla en 12 columnas con filas. Con el wrapper permite definir cada tamaño de columnas y diferentes colores y formas más o menos estándar.

En angular.json se define la hoja de estilos de toda la aplicación. Una vez instalado bootstrap, podemos poner en "styles" la hoja de estilos de bootstrap: bootstrap /dist/css/min

Un módulo es como un componente pero sirve para establecer toda la seguridad (quien puede usar) nuestros componentes.

declarations qué modulo usar

imports - librerías u otros

exports - se suele usar para módulos que se exportan a otro lado

providers - librerías a usar

En W3S se pueden encontrar mucha info sobre las opciones de estilos que nos proporciona Bootstrap

# ¿Qué puedo hacer con Angular?

## Eventos de Angular

(clic), (change), (summit), (blur) son los más habituales

Sirven para asociar un componente del DOM con un evento y así tener algo de lógica.

Angular ya enlaza el HTML con el componente, de forma que ya está enlazado el evento con su función

Ejemplo, al hacer clic en una cabecera, saldrá un alert:

<h1 (click)="clickHeader()">mi cabecera</h1>

Se pueden concatenar varias funciones e incluso asignar variables en el propio html. Aunque no es recomendable.

La función se mete en app.component.ts. El código html recordar que está en app.component.html

Dentro de cualquier página podemos llamar a un selector cualquier para reutilizar el código de otro componente.

Para pintar una variable directamente en HTML alert (`Hola ${this.miCadena}`);

Se tienen que usar las comillas hacia la izquierda.

Clic y blur son útiles en pareja para los eventos de seleccionar y luego con el blur cuando el usuario termina la acción.

# Data binding

Sirve para vincular variables en las plantillas HTML.

## Interpolation

Pintar el valor de una variable en HTML

{{nombreVariable}} directamente escrito en el HTML.

De hecho, al cambiar desde el código, cambiará en el HTML.

## Property binding

Asociar el valor de una propiedad HTML con el de una variable en código. Así funcionará dinámicamente:

<input [value]=”name” />

Y para que el usuario pueda cambiar directamente los valores de nuestras variables, tenemos que importar en el módulo FormsModule

## Event binding

Se asocia con una función: (clic)=”lanzafuncion()”

## Two-way binding

Permite asociar el valor de una variable en mi código con el de un value en HTML.

E incluso con una propiedad de una clase

<input[(ngModel)] = “name” />

<input[(ngModel)] = “alumno.nombre” />

# Práctica

Tres botones asociados a un evento clic, que por detrás haga un random y comparar con lo hecho por el usuario.

# Variables en TypeScript

const para definir constantes

let para definir variables

Cuando interpolemos las cadenas en el código, hay que usar comillas hacia la izquierda:

Alert(`Hola ${miVariable}`)

# Directivas más utilizadas en Angular

Ngfor sirve para iterar

\*ngFor = “let nombreVariable of nombreArray; let nombreIndice = index

<p (clic)=”nombreVariable.metodo()”>

{{nombreVariable.atributo}}

Ngif elimina el contenido del DOM, así que es más seguro que un hidden

<p \*ngIf=”variable”>Este párrafo aparece según condición</p>

[hidden] no es estructural y sólo oculta en la interfaz, pero no elimina

Otras no estructurales no tan importantes:

[ngClass], [ngStyle], [ngSwitch]

# Segundo día: trabajo con rutas

Además, vamos a hacer un conversor de temperaturas entre tres, de manera automática al escribir entre los diferentes campos.

# Rutas

La herramienta es el Router de Angular.

Permite

* Definir rutas incluso navegables
* También sirve para registrar la actividad.
* Y permite además pasar parámetros entre cada una.

1. Hay que importar un nuevo módulo

Ng generate module app-routing

Crea un nuevo modulo app-routing-module.ts:

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

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

const routes: Routes = [];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

Los exports son para que todo el mundo tenga acceso, porque para eso es para lo que sirven las rutas.

Los 4 imports son necesario siempre y así se hace, no hay más tutía.

## Una ruta es un objeto JSON

const routes: Routes = [

  {

    path: 'home',

    component: AppComponent

  },

  {

    path: '',

    redirectTo: '/home'

  }

];

#### Otras opciones

* Las opciones can son para dar permisos para acceder y otras cosas
* Redirectto es para redireccionar
* pathMatch: ‘full’ es para que la ruta sólo se entienda exactamente con ese nombre, ya que el navegador de rutas va a intentar ir comparando de arriba abajo. (es como el false en el vbuscar de Excel).
* El comodín, por esto último hay que ponerlo al final:
  + Path: ‘\*\*’,Component: PageNotFoundComponent

#### Paso de parámetros

Path: ‘example/:miVariable’

## Directivas para usar las rutas

<router-outlet></router-outlet> en el app.component

Para redirigir:

<a routerLink=”/home”

Opciones

routerLinkActive permite asociarle una clase al componente

En Bootstrap tenemos también una clase llamada offset para crear columnas vacías a la izquierda.

# Personalizar Angular para ahorrarnos trabajo

Crear una palabra clave para que en un elemento haga esa lógica, para reutilizar código. Para cambiar.

Ng generate directive nombreDirectiva

Se declara en el módulo principal

Se usa donde queremos. Por ejemplo:

Constructor(private elemento **: ElementoRef**)

# Decoradores de angular

#### De componente y módulo

@algo encima de una clase.

#### Decoradores de propiedad

@Input

@Output

#### Decoradores de método

@HostListener : escucha eventos (por ejemplo el de input).

#### De parámetros

@Inject : sirve para usar el parámetro solo en una clase.

@Injectable : para crear Singleton

En un método, se pueden poner parámetros opciones: funcion(color?: string)

Para evitar que se aumente el número de directivas en las declaraciones de app.module, lo que se hace es crear un nuevo módulo para organizar ahí todas las directivas

# Pipes

Una forma de modificar el valor que estamos pintando (interpolando) en el front. Una especie de formateo. Date, Currency, LowerCase, UpperCase, TitleCase, Decimal, Json

Ejemplo: {{valueDate | date: ‘’dd/MM/yyyy’}}

Además, de esos estándar de Ángular, se pueden crear propios.

# Formularios

Vamos a usar el Two-way binding (ngmodel).

Validación de inputs y gestión de errores.

Cada formulario un nuevo componente. En HTML será <form>, gracias al módulo importado FormsModule

La etiqueta en HTML la asignaremos al componente así: <form #miFormulario = “ngForm”></form>

El formulario creará un nuevo objeto.

Con eso en el HTML, Bootstrap proporciona una clase para dar uniformidad a los campos del grupo: class="form-group"

En los input se puede usar una clase class="form-control"

Para poder acceder a los formularios hay que asignarlos: <form (ngSubmit)="summit()" #miPrimerFormulario="ngForm">

Para poder acceder a los inputs hay que asignar los nombres: <input #nombre="ngModel"

## Estado de los inputs con ngmodel

* Ng-touched/ng-untouched : clicado
* Ng-dirty/ng-pristine : cambiado
* Ng-valid/ng-invalid : ha pasado la validación establecida

En el CSS podemos hacer este ejemplo.

* .ng-valid[required], .ng-valid.required {
* border-left: 5px solid green;
* }
* .ng-invalid:not(form) {
* border-left: 5px solid red;
* }

La primera opción tiene dos valores porque el primero es para el valor del Angular y el segundo es por si tiene algún hijo required.

En la segunda queremos que sólo se aplique a los campos y no al formulario.

Para acceder a nuestro formulario desde el TS

1. Mediante FormBuilder desde el TS
2. Con el decorador @ViewChild(), que decora una variable:

@ViewChild('miFormulario', {static: false}) public miFormulario: NgForm;

### Directivas para validación

* Required
* requiredTrue : normalmente para checkBox
* minLength/maxLength
* min/max: para los input numéricos. Pero solo es un limitador, no un validador
* pattern: para controlar expresiones regulares
* email : igual que pattern pero para email.
* Y luego las nuestras propias: se crean directivas implementando la interfaz Validator
  + NG\_Validators nos proporciona unas cuantas opciones de validación ya estándar.
  + Al crear una directiva y validar, en el caso de crear una función fuera hay que crear una función que devuelve una función:

      return this.isMinValue(this.minValue)(formControl);

    }

  }

  isMinValue(minValue : number) {

    return (formControl : FormControl) => {

      const valorControl = formControl.value;

      if(!valorControl && valorControl != 0 || isNaN(valorControl)) return null;

      return valorControl < minValue ? { minValue:true } : null;

    }

  }

* + LA directiva se puede aplicar en el formulario o en cada campo:

[appMinValue] = "16"

* + La directiva en el HTML, si tiene parámetros tiene que ir entre corchetes:

 [appMustMatch]="['email', 'confEmail']"

En las validaciones en HTML podemos acceder a los errores dados en la validación usando \*ngIf y comprobando dentro del campo, el atributo errores y dentro el tipo de validación:

<small \*ngIf="nombre.errors && !nombre.errors.required && nombre.errors.minLength" id="nombreHelpMinLength" class="form-text text-muted alert alert-danger">¡El nombre requiere al menos 3 caracteres

                    !</small>

# Clases

Vamos a crear una carpeta models (el nombre estándar) donde crearemos las clases.

Y creamos una clase que no necesita más que usarse:

export class User {

  id?: number;

  name?: string;

  surname?: string;

  age?: number;

  dni?: string;

  email?: string;

  constructor(id?: number, name?: string, surname?: string, age?: number, dni?: string, email?: string) {

    this.id = id || null;

    this.name = name || null;

    this.surname = surname || null;

    this.age = age || null;

    this.dni = dni || null;

    this.email = email || null;

  }

  toString() {

    console.log(`Usuario: ${this.name} ${this.surname}`);

  }

}

Además de las clases, se pueden crear interfaces (durante la creación de la misma clase).

Export interface UserInterface {

Id?: number;

Nombre: string;

toString();

}

Se usa en la clase “implements UserInterface”

# Servicios

Lo correcto para trabajar con peticiones HTTP y almacenar datos únicos (es como una especie de sesión entre componentes). También se puede usar para almacenar la lógica a compartir en toda la aplicación, por ejemplo, la comprobación de validez del NIF.

Se trata de instancia singleton

Ng generate service myService

Lo podemos meter en el constructor de nuestro componente para que lo pueda usar:

constructor(private userService : UserService)

# Observables

Las promesas en javascript es para esperar eventos que terminan (asíncronos).

En TS, ahora se trabaja con Observables que nos permiten establecer un método que nos sea llamado cuando ocurra algo en otro objeto. Y para gestionarlo, lo mejor es usar las suscripciones (las promesas eran más enrevesadas y los pipes son para transformar).

  ngOnInit(): void {

    this.userService.getStudents().subscribe(students => {

      this.users = students;

    });

Hay que tener cuidado porque la suscripción no es síncrona, así que se reciben los datos en cualquier momento. Esto quiere decir que el flujo continua sin haber realizado la instrucción interna.

Y en el servicio, hay que declararlo así, para que devuelva un Observable con los datos:

  getStudents(): Observable<User[]> {

    return of(this.students);

  }

También nos podemos unsubscribe de lo pedido. Es muy común cuando cambia algo en el cliente y se tiene que renovar la suscripción.

# HTTP en Angular

Un módulo a importar en app.module (HttpClientModule).

Para usarlas en un servicio, utilizar la dependencia HttpClient, en el constructor como referencia privada.

Con esto, nos permite acceder a los métodos habituales. GET POST PUT DELETE.

Y luego apuntar al endpoint (url). Lo mejor usando observables desde el componente. También podemos usar los pipes para cambiar la salida y así por ejemplo manejar los errores.

### Errores HTTP

Angular nos devuelve un código de error en HttpErrorResponse y así podemos usar un segundo método.

Otra forma es lidiar con ellos mediante pipe, que queda más limpio. Así tendrá un método específico, comprobamos de donde viene y podemos dar respuestas específicas.

return this.http.get<Post[]>(`${this.resourceUrl}/${id}`, { observe: 'response' }).pipe(catchError(this.handleError));

<https://jsonplaceholder.typicode.com/> para ver los ejemplos de llamadas.

() -> emite

[] recibe

[()] -> recibe y emite

Environments

Angular permite diferentes configuraciones para entornos como desarrollo o producción o también por ejemplo para dos países diferentes.

Es cuando la aplicación funciona prácticamente igual salvo ciertas variables o algunos comportamientos del entorno.

Carpeta environments. Angular.json para declarar las nuevas configuraciones ("configurations")

Por defecto, Angular tiene un entorno creado de producción ya creado por defecto.

#### Para crear el entorno de producción, que hace que funcione más ligero

Ng serve -- prod

Ng build –prod

# GIT

Instalar (ver opciones de instalación en: <https://www.atlassian.com/es/git/tutorials/install-git>)

Git init

git add . para añadir todo el proyecto

En .gitignore se pueden añadir los ficheros y carpetas que no queramos subir a git

Git commit almacenará los cambios en local. Al hacerlo, arriba del todo hay que escribir el mensaje del commit

Ahora tenemos que conectar al repositorio

git remote add origin <https://github.com/adriftag/cursoangular.git>

Si ya tuviéramos un repositorio, para cambiarlo hay que ejecutarlo con la opción remove origin

git push -u origin master

Git push subirá los cambios al repositorio.

### Github desktop

Además de Github online, podemos tenerlo en local.

Lo bueno es que escucha todos los cambios que se realicen en el proyecto y te los muestra disponible ya directamente para subir. Una vez subido en local, nos avisará para hacer un push al servidor.

De la línea de seguridad master, se saca un branch de develop principal. A partir de un punto (Versión), se sacan ramas para trabajar. Una vez alguien haga push al branch de desarrollo, la persona que sacó su rama, tendrá que hacer un pull previo para mezclar con su rama y luego ya podrá hacer el push sin problemas.

Así que enseguida debemos crear y publicar la rama develop.

Los conflictos al mezclar se manejan mejor desde Visual Studio Dev, pero también se pueden ver en Github Desktop.

Sourcetree es otro cliente de Git que se integra muy bien con Jira.

# Lazy Loading

Creamos módulos aparte para las rutas y para los componentes

ng generate module post

ng generate module postRouter

Movemos todos los componentes de post al nuevo módulo Post

La única diferencia con el módulo enrutador principal es esta:

imports: [RouterModule.forChild(routes)]

En el app-routing se carga las rutas del nuevo módulo mediante una complicada sintaxis:

loadChildren: () => import('./post/post.module').then(mod => mod.PostModule)

Vamos a crear un modulo para compartir todo entre todos:

ng generate module shared

Aquí hay que exportar así:

export class SharedModule {

  static forRoot() {

    return { ngModule : SharedModule

    };

  }

}

Luego, en app-module hay que quitar todos los imports salvo:

  imports: [

    BrowserModule,

    AppRoutingModule,

    SharedModule.forRoot()

  ],

Y en los demás módulos existentes, basta con importar el SharedModule:

  imports: [

    PostRouterModule,

    SharedModule

  ]

# Router Guards

Para limitar entrada o salida a diferentes rutas. Condiciones como estar LOGUEADO o al terminar de completar una formulario, etc.

Devuelve true, false o un árbol de URLs que cancelan la navegación para reiniciarla en otro lugar.

Normalmente trabajan con Obervables para poder hacer consultas pesadas y por eso hacerlo asíncronamente

Pueden usar diferentes interfaces:

* canActivate
* canActivateChild
* canDeactivate
* …

ng generate guard auth

En la ruta que queramos proteger:

canActivate: [AuthGuard]

# Documentación

<http://www.github.com/eduardo-garcia-melit/curso-febrero>

npm install sfsfsdfslibreria --save

Siempre usar el último parámetro para que se guarde la versión y así funcione la resolución de dependencias.

<https://angular.io/guide/styleguide>

Libro Clean Code también para Java