Contenido Procesado

1. servicio: 35 veces
2. usar: 17 veces
3. sistema: 16 veces
4. from: 16 veces
5. angular: 14 veces
6. import: 13 veces
7. componente: 13 veces
8. servicios: 11 veces
9. módulo: 11 veces
10. array: 11 veces

DWEC06.- angular y TypeScript II

En esta UD vamos a trabajar la última parte referente al Framework de desarrollo angular. En ella trabajaremos las funciones necesarias para que podáis desarrollar una aplicación web completa con las funcionalidades básicas que debería tener toda aplicación, haciendo especial hincapié en los servicios para que la web se alimente y pueda gestionar la información de una Base de Datos a través de una APIs y sus correspondientes consultas HTTP.

1.- Routing

El sistema de routing no es un asunto trivial, ya que envuelve a muchas clases, objetos, componentes y configuraciones. De hecho es algo bastante sofisticado, ya que tiene decenas de configuraciones para realizar rutas de todo tipo.

Pensando en aquellas personas que comienzan con angular su incursión en las "SPA" (Single Page Application), comenzaremos aclarando qué es un sistema de routing y por qué lo necesitamos.

En cualquier sitio web generalmente tienes varias direcciones que son entregadas por un servidor, para mostrar diferentes contenidos del sitio. Podemos tener una portada, una página de productos, una de contacto, etc. Cada una de esas páginas se presenta en una ruta diferente del sitio web, que podrían ser como *example.com, example.com/productos*

/index.html, example.com/contacto.html, etc. Cada una de esas rutas podría tener un archivo HTML, que se sirve con el contenido de esa sección. Hasta aquí estamos seguros que los lectores no tendrán ningún problema, pues es así como funcionan, en líneas generales, prácticamente todos los sitios web.

example.com, example.com/productos/index.html, example.com/contacto.html,

Para facilitar la navegación por un sitio donde realmente sólo hay un index, existe lo que llamamos **el sistema de routing**, que tiene el objetivo de permitir que en el sitio web haya rutas internas, **respondiendo a rutas "virtuales"** como las que existen en los sitios tradicionales.

Llamamos "virtuales" a esas rutas, porque realmente sólo existe un "index.html", no habrá un archivo "contacto.html" o "productos.html" para cada ruta, sino que será realmente siempre el "index.html" el que se entregue al navegador.

El sistema de routing es el encargado de reconocer cuál es la ruta que el usuario quiere mostrar, presentando la pantalla correcta en cada momento. Esto es útil por varios motivos, entre ellos:

- Permite que la aplicación responda a rutas internas. Es decir, no hace falta entrar siempre en la pantalla principal de la aplicación y navegar hasta la pantalla que queremos ver realmente.

Permite que la aplicación responda a rutas internas. Es decir, no hace falta entrar siempre en la pantalla principal de la aplicación y navegar hasta la pantalla que queremos ver realmente.

- Permite que el usuario pueda usar el historial de navegación, yendo hacia atrás y hacia adelante con el navegador para volver a una de las pantallas de aplicación que estaba viendo antes.

Permite que el usuario pueda usar el historial de navegación, yendo hacia atrás y hacia adelante con el navegador para volver a una de las pantallas de aplicación que estaba viendo antes.

1.1.- El sistema de Routing en angular

angular, como un buen framework, dispone de un potente sistema de routing para facilitar toda la operativa de las single page applications. Está compuesto por varios actores que tienen que trabajar juntos para conseguir los objetivos planteados.

Comenzaremos explicando el sistema de routing resumiendo los elementos básicos que forman parte de él y que son necesarios para comenzar a trabajar.

- El módulo del sistema de rutas: llamado **RouterModule**.

El módulo del sistema de rutas: llamado **RouterModule**.

- Rutas de la aplicación: es un array con un listado de rutas que nuestra aplicación soportará.

Rutas de la aplicación: es un array con un listado de rutas que nuestra aplicación soportará.

- Enlaces de navegación: son enlaces HTML en los que incluiremos una directiva para indicar que deben funcionar usando el sistema de routing.

Enlaces de navegación: son enlaces HTML en los que incluiremos una directiva para indicar que deben funcionar usando el sistema de routing.

- Contenedor: donde colocar las pantallas de cada ruta. Cada pantalla será representada por un componente.

Contenedor: donde colocar las pantallas de cada ruta. Cada pantalla será representada por un componente.

1.- Importar el código del sistema de routing

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

RouterModule es el módulo donde está el sistema de rutas, por tanto contiene el código del propio sistema de rutas.

Routes es una declaración de un tipo, que corresponde con un array de objetos Route. Los objetos Route están declarados por medio de una interfaz en el sistema de routing. Esta interfaz sirve para que en la declaración de tus rutas coloques solamente aquellos valores que realmente son posibles de colocar. Si no lo haces así, el compilador de TypeScript te ayudará mostrando los correspondientes errores en el tiempo de desarrollo.

2. Crear la lista de las rutas disponibles en la aplicación

En este paso tenemos que crear un array de las rutas que queremos generar en nuestra aplicación. Utilizaremos la declaración del tipo Routes, que hemos dicho es un array de objetos Route.

Cada objeto Route del array tiene un conjunto de campos para definir la ruta, definido por la interfaz Route, siendo que lo general es que tenga al menos un camino "path" y un componente para representar en esa ruta.

El código que producirá será más o menos así.

```
const rutas: Routes = [

  { path: '', component: HomeComponent },
```

```
  { path: 'contacto', component: ContactoComponent }
];
```

3.- Declarar el sistema de routing en el "imports" del decorador @NgModule

Ahora hay que editar el decorador del módulo principal @NgModule, donde tendrás que colocar en el array "imports" la declaración de que vas a usar el sistema de routing de angular.

Tenemos que indicar a RouterModule, pero en ese imports necesita la configuración de rutas, creada en el array Routes[] del paso anterior. De momento, esa configuración la cargamos mediante un método forRoot() en el que pasamos por parámetro el array de rutas.

Se ve mejor con el código. Este es el decorador completo, pero te tienes que fijar especialmente en el array "imports".

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ContactoComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(rutas)
  providers: [],
  bootstrap: [AppComponent]
})
```

4.- Definir el template del componente principal

Unavez hecho esto, comenzamos el trabajo en el componente principal. En este componente tenemos que meter un poco de código en la vista, archivo "app.component.html".

En el template se crea el navegador de las rutas, indicando en los enlaces la directiva "routerLink", con el valor de cada ruta.

```
<nav>
  <a routerLink="/">Home</a> |
  <a routerLink="/contacto">Contacto</a> |
```

```
</nav>
```

2.-servicios

Básicamente hemos dicho anteriormente que el protagonista en las aplicaciones de angular es el componente, que las aplicaciones se desarrollan en base a un árbol de componentes. Sin embargo, a medida que nuestros objetivos sean más y más complejos, lo normal es que el código de los componentes también vaya aumentando, implementando mucha lógica del modelo de negocio.

En principio esto no sería tan problemático, pero sabemos que la organización del código, manteniendo piezas pequeñas de responsabilidad reducida, es siempre muy positiva. Además, siempre llegará el momento en el que dos o más componentes tengan que acceder a los mismos datos y hacer operaciones similares con ellos, que podrían obligarnos a repetir código. Para solucionar estas situaciones tenemos a los servicios.

Básicamente un servicio es un proveedor de datos, que mantiene lógica de acceso a ellos y operativa relacionada con el negocio y las cosas que se hacen con los datos dentro de una aplicación. Los servicios serán consumidos por los componentes, que delegarán en ellos la responsabilidad de acceder a la información y la realización de operaciones con los datos.

2.1.- Crear un servicio

Tal como viene siendo costumbre en el desarrollo con angular, nos apoyaremos en angular CLI para la creación del esqueleto, o scaffolding, de un servicio.
Para crear un servicio usamos el comando "generate service", indicando a continuación el nombre del servicio que queremos generar.

```
ng generate service clientes
```

Esto nos generaría el servicio llamado "ClientesService". La coletilla "Service", al final del nombre, lo agrega angular CLI, así como también nombra al archivo generado con la finalización "-service", para dejar bien claro que es un servicio.

Es habitual que quieras colocar el servicio dentro de un módulo en concreto, para lo que puedes indicar el nombre del módulo, una barra "/" y el nombre del servicio. Pero atención: ahora lo detallaremos mejor, pero queremos advertir ya que esto no agregará el servicio al código de un módulo concreto, sino que colocará el archivo en el directorio de ese módulo. Enseguida veremos qué tienes que hacer para que este servicio se asigne realmente al módulo que desees.

```
ng generate service facturacion/clientes
```

1. Agregar la declaración del servicio a un módulo

Para poder usar este servicio es necesario que lo agregues a un módulo. Inmediatamente lo podrás usar en cualquiera de los componentes que pertenecen a este módulo. Este paso es importante que lo hagas, puesto que, al contrario de lo que ocurría al crear un componente con el CLI, la creación de un servicio no incluye la modificación del módulo donde lo has creado.

Así pues, vamos a tener que declarar el servicio manualmente en el módulo. Lo haremos gracias al decorador del módulo (@NgModule), en el array de "providers" e importarlo en su correspondiente módulo. El decorador de un módulo con el array de providers podría quedar más o menos así.

```
@NgModule({
 imports: [
 CommonModule
 ],
 declarations: [ListadoClientesComponent],
 providers: [ClientesService]
})


import { ClientesService } from './clientes.service';
```

2. Código básico de un service en angular

Ahora podemos examinar el código generado para nuestro servicio y aprender nuevas cosas de angular. Este sería nuestro recién creado servicio "ClientesService"

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

@Injectable()
export class ClientesService {

constructor() { }

}
```

Como verás, el servicio no tiene nada todavía, solo su declaración, pero hay cosas interesantes que tenemos que explicar, principalmente un nuevo decorador que no habíamos conocido hasta el momento: "@injectable".

El decorador @injectable indica a angular que la clase que se decora, en este caso la clase ClientesService, puede necesitar dependencias que puedan ser entregadas por inyección de dependencias. De momento puedes quedarte que los servicios necesitan de este decorador, aunque realmente disponer de él no es condición indispensable.

El import, arriba del todo, { Injectable } from '@angular/core', lo que hace es que nuestra clase conozca y sea capaz de usar el decorador @injectable.

Por lo demás, el servicio está vacío, pero le podemos poner ya algo de código para que

nos sirva de algo. De momento vamos a exponer mediante el servicio una simple propiedad con un dato, que luego vamos a poder consumir desde algún componente. Para ello usamos las propiedades de la clase, tal como nos permite TypeScript.

```
export class ClientesService {
 accesoFacturacion = 'https://login.example.com';
 constructor() { }
}
```

En nuestra clase ClientesService hemos creado una propiedad llamada "accesoFacturacion", en la que hemos asignado un valor que sería común para todos los clientes. El dato es lo de menos, lo interesante es ver que la declaración no dista de otras declaraciones en clases que hayamos visto ya.

Debes tener en cuanta una cosa muy importante. A la hora de crear el servicio, mediante el decorador Injectable, estamos utilizando un patrón de programación Singleton, de modo que solo se crea una instancia del objeto del modelo de datos, y así esta puede ser llamada desde cualquier componente y utilizar la misma información

2.2.- Uso del servicio

1. Cómo inyectar dependencias de servicios

Ahora nos toca ver la magia de angular y su inyección de dependencias, que vamos a usar para poder disponer del servicio en un componente.

Como en otros frameworks, en angular la inyección de dependencias se realiza por medio del constructor. En el constructor de un componente, que hasta ahora habíamos dejado siempre vacío, podemos declarar cualquiera de los servicios que vamos a usar y el framework se encargará de proporcionarlo, sin que tengamos que realizar nosotros ningún trabajo adicional.

Esto es tan sencillo como declarar como parámetro la dependencia en el constructor del componente.

```
constructor(public clientesService: ClientesService) { }
```

De esta manera estamos indicando a TypeScript y angular que vamos a usar un objeto "clientesService" que es de la clase "ClientesService". A partir de entonces, dentro del componente existirá ese objeto, proporcionando todos los datos y funcionalidad definida en el servicio

2. Usando el servicio en el componente

Ahora, para acabar esta introducción a los servicios en angular, tenemos que ver cómo usaríamos este servicio en el componente. No nos vamos a detener demasido en hacer ejemplos elaborados, que podemos abordar más adelante, solo veremos un par de

muestras sobre cómo usar la propiedad declarada en el servicio.

1.- Usando el servicio dentro de la clase del componente

Dentro de la clase de nuestro componente, tendremos el servicio a partir de la propiedad usada en su declaración. En el constructor dijimos que el servicio se llamaba "clientesService", con la primera en minúscula por ser un objeto.

Pues como cualquier otra propiedad, accederemos a ella mediante la variable "this".

```
export class ListadoClientesComponent implements OnInit {

ngOnInit() {
 console.log(this.clientesService);
 }

}
```

El ejemplo no vale para mucho, solo para mostrar que, desde que esté creado el objeto podemos acceder al servicio con "this.clientesService".

También nos sirve para recordar que, el primer sitio donde podríamos usar los servicios declarados es en el método ngOnInit(). Dicho de otro modo, si necesitamos de estos servicios para inicializar propiedadades en el componente, el lugar donde ponerlos en marcha sería el ngOnInit().

2.- Usando el servicio en el template de un componente

Por su parte, en el template de un componente, podrás también acceder al servicio, para mostrar sus propiedades o incluso invocar sus métodos como respuesta a un evento, por ejemplo.

Como el servicio está en una propiedad del componente, podremos acceder a él meditante ese nombre de propiedad. Pero ten en cuenta, que este acceso desde el template sólo será posible si el servicio se declaró con visibilidad "public".

```
<p>
 URL De acceso: {{clientesService.accesoFacturacion}}
</p>
```

2.3.- Consultas HTTP

Llamadas HTTP a una API REST - ¿Cómo usar HTTPClient?

Hasta ahora con lo que sabemos podemos usar los servicios para leer y escribir datos, pero simulados o guardados en la memoria del navegador, para leer o escribir de una API REST tenemos que hacer llamadas HTTP.

Antes de empezar con las llamadas hay que conocer que angular tiene un modulo que

facilita esta tarea, el modulo es HttpClient. Con este modulo no necesitas usar fetch ni ajax ni nada.

Para usar HttpClient de angular en cualquier parte, tenemos que importar el módulo HttpClientModule, en la sección imports de el app.module.ts:

```
import { HttpClientModule } from "@angular/common/http";
```

Aunque lo hayamos importado en la página de forma global, también tenemos que importarlo y inyectarlo en los constructores de los servicios desde los que vayamos a realizar llamadas HTTP. Por ejemplo para este nuevo servicio que he creado para listar los usuarios de que vienen de la API:

```
import { HttpClient } from "@angular/common/http";

@Injectable({
 providedIn: "root"
})
export class UsersService {
 constructor(private http: HttpClient) {}
}
```

HttpClient usa Observables de RxJS. Los observables son una colección de futuros eventos que llegan de forma asíncrona. Si quieres aprender más de RxJS puedes visitar su web oficial

Con eso listo ya puedes realizar llamadas HTTP:

- GET: Simplemente devuelven información.

GET: Simplemente devuelven información.

- POST: A estos endpoints se envía información normalmente para crear o ejecutar acciones sobre recursos en bases de datos.

POST: A estos endpoints se envía información normalmente para crear o ejecutar acciones sobre recursos en bases de datos.

- PUT: Se envía información al endpoint y se modifica en base de datos un recurso.

PUT: Se envía información al endpoint y se modifica en base de datos un recurso.

- DELETE: Para borrar recursos del servidor.

DELETE: Para borrar recursos del servidor.

Ahora para realizar llamadas http podemos invocar cualquiera de los siguiente métodos definidos en el HttpClient:

- get()

get()

- post()

post()

- put()

put()

- delete()

delete()

2.3.1.- Llamada GET

Son las llamadas más básicas y como hemos dicho sirven para devolver información desde el servidor. Esta información puede ser un dato simple, un objeto o una lista de objetos todo ello en formato JSON normalmente.

Por ejemplo para hacer una llamada get a la API de ejemplo que hemos comentado antes:

```
getUsers(){
  this.http.get('https://reqres.in/api/users?page=2').subscribe(data => {
    console.log(data);
  });
  console.log("Esto se ejecutará antes que el console log de arriba");
}
```

Fíjate que ahora después de la llamada al método get() usamos subscribe. Esto funciona como las promesas de Javascript, con ese método te esperas a que la petición termine. Dentro de ese método se ejecuta una función (usando arrow functions/callback) en la que se devuelve el objeto data que contiene la respuesta a la petición de la API.

El console.log de abajo se ejecuta antes incluso de que termine la petición. Esto pasa porque el código es asíncrono y por tanto lo que pongas debajo no va a esperar a que la petición termine. Dentro del subscribe si que tienes la certeza de que la petición ha terminado y por tanto tienes la respuesta.

El servicio completo quedaría así:

```
import { HttpClient } from "@angular/common/http";


@Injectable({
  providedIn: "root"
})
export class UsersService {
  constructor(private http: HttpClient) {}
```

```
  getUsers() {
    this.http.get("https://reqres.in/api/users?page=2").subscribe(data => {
      console.log(data);
    });
  }
```

Ahora podemos inyectar este servicio como ya hemos hecho, en este caso en el componente de lista de usuarios:

```
import { Component, OnInit } from "@angular/core";
import { UsersService } from "./users.service";
@Component({
  selector: "app-users",
  templateUrl: "./users.component.html",
  styleUrls: ["./users.component.css"]
export class UsersComponent implements OnInit {
  constructor(public usersService: UsersService) {}
  ngOnInit() {
    this.usersService.getUsers();
```

Con esto podemos realizar llamadas desde el servicio pero los datos que se devuelven todavía no los tenemos en el componente para poder mostrarlos.

Para hacer poder mostrar los datos en el HTML del componente, en lugar de hacer el subscribe en el servicio, tenemos que devolver un Observable de la llamada http, es decir:

```
getUsers(): Observable<any>{
  return this.http.get('https://reqres.in/api/users?page=2');
```

No sin antes importar los observables en el servicio:

```
import { Observable } from "rxjs/Observable";
```

Para este ejemplo he puesto que el Observable sea de tipo Any (cualquier tipo de objeto) pero lo suyo, en un futuro, sería usar una interfaz para poder tener un modelo para los datos.

Ahora, en el componente, cuando queramos llamar al servicio (siempre y cuando lo hayamos inyectado en el controlador), tenemos que subscribirnos para recibir la información, es decir:

```
import { Component, OnInit } from "@angular/core";

import { UsersService } from "../users.service";

  users: any;

    this.usersService.getUsers().subscribe(data => {

      this.users = data;
```

IMPORTANTE *Cuando carga un componente, si mostramos una variable que viene de una petición HTTP, no se cargará y tirará error porque en el instante en el que se abre la página, la petición aún no se ha realizado. Para arreglar esto tenemos que poner un ngIf a la variable que viene desde la petición antes de mostrarla en la vista.*

IMPORTANTE *Cuando carga un componente, si mostramos una variable que viene de una petición HTTP, no se cargará y tirará error porque en el instante en el que se abre la página, la petición aún no se ha realizado. Para arreglar esto tenemos que poner un ngIf a la variable que viene desde la petición antes de mostrarla en la vista.*

```
<div *ngIf="users">

  {{ users }}

</div>
```

2.3.2.- Llamadas POST y PUT

Las llamadas POST y PUT sirven para enviar información al servidor y que éste nos responda. POST se usa para crear recursos, por ejemplo, crear usuarios, crear artículos o lo que sea. Normalmente se pasa un objeto o conjunto de objetos a crear. Las peticiones PUT sirven para actualizar un objeto ya creado. Se pasa lo mismo que en el post, un objeto o conjunto de ellos para editar por los que haya se hayan creado.

Para conseguir esto simplemente al realizar la llamada que corresponda, desde el servicio pasamos el objeto correspondiente, por ejemplo:

```
createUser(user: Any): Observable<any>{
 return this.http.post('https://reqres.in/api/users', user);
}
editUser(user: Any): Observable<any>{
 return this.http.post('https://reqres.in/api/users/2', user);
}
```

El segundo parámetro de la función del get y del post es el objeto o objetos que quieres enviar al servidor.

Con esto ya podemos usarlo dentro del componente de users:

```
import { Component, OnInit } from "@angular/core";
import { UsersService } from "../users.service";

@Component({
```

```
 selector: "app-users",
 templateUrl: "./users.component.html",
 styleUrls: ["./users.component.css"]
})
export class UsersComponent implements OnInit {
 users: any;

constructor(public usersService: UsersService) {}

ngOnInit() {
 this.usersService.createUser({
 name: "morpheus",
 job: "leader"
 });
 this.usersService.editUser({
 name: "morpheus",
 job: "zion resident"
 });
 }
}
```

Por cierto, en los post y los put también puedes hacer un subscribe para esperar a que termine la petición por si quieres ejecutar algo después o recibir la información del servidor, la sintaxis es la misma que con el GET.

2.3.3.- Llamada DELETE

La llamada DELETE se usa para borrar recursos del servidor y si suntaxis es exactamente igual que las que hemos visto anteriormente:

```
deleteUser(): Observable<any>{
 return this.http.delete('https://reqres.in/api/users/2');
}
```

2.3.4.- Envío de Headers

Con lo que hemos visto ahora te sirve para usar muchas APIS, pero puede darse el caso en el que tu API esté protegida con un sistema de Basic auth. En ese caso, para no tener que pasar esa información en cada petición lo que puedes hacer es extender HttpClient para pasar estos headers en todas las peticiones sin que te tengas que acordar de hacerlo cada vez.

Para ello, una de las soluciones posibles es crear un archivo .ts para pasar en todas las llamadas los headers:

```
import { Injectable, OnInit } from "@angular/core";
import { Http, Headers, RequestOptions } from "@angular/http";

@Injectable()
export class HttpWithHeaders {
 public sessionData: SessionData;

constructor(private http: Http) {
 this.headers = "Basic " + btoa("username" + ":" + "password");
 }
 get(url) {
```

```
 return this.http.get(url, {
 headers: this.generateHeaders()
 });
 }
 post(url, data) {
 return this.http.post(url, data, {
 headers: this.generateHeaders()
 });
 }
 put(url, data) {
 return this.http.put(url, data, {
 headers: this.generateHeaders()
 });
 }
 delete(url) {
 return this.http.delete(url, {
 headers: this.generateHeaders()
 });
 }
}
```

Para usar esta clase que acabamos de crear, simplemente tenemos que cambiar en los servicios el httpClient por nuestro HttpWithHeaders.