**Angular 8**

Al instalar el proyecto se generan una serie de carpetas, además nosotros necesitamos organizar los distintos tipos de ficheros que vamos a ir generando en el desarrollo.

**Ficheros de angular**

* index.html: Es el primer fichero que se ejecuta, contiene la cabecera con las importaciones e inicia el contenido principal de la aplicación.
* styles.scss: Los estilos principales de la aplicación.
* app/app.component.ts: El primer componente que se ejecuta tras el index.html
* app/app.module.ts: Indica al sistema el conjunto de ficheros que se tendrán en cuenta en la aplicación y deben estar disponibles.

**Organización de carpetas por tipos**

* 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: Incluiremos los ficheros para tratar con servicios web, pero también los que necesitemos para interconectar otros componentes o extraer código. Este tipo de ficheros (services) serán los que contengan la lógica de la aplicación.
* Views: Para mejorar la clasificación de ficheros, aquí deberías incluir todos aquellos componentes que componen las vistas de nuestra aplicación, y en ellos importar nuestros componentes.
* Menu: Al igual que las vistas, es una buena opción incluir aquí los componentes que componen el menú de la aplicación.

En angular 8 usaremos marcas en el html, todas estas marcas son directivas y nos permiten interactuar con los “templates”. A continuación veremos varias de ellas.

**Directivas que encontraremos**

- {{ variable }} : Nos sirve para mostrar el contenido de una variable. - [(ngModel)] : Para enlazar variables a elementos del DOM, como a un input. - ( event ) : Para capturar y asignar a funciones eventos disparados en los templates, como el evento click, básico para cualquier objetivo. - <app-component></app-component> : Las directivas para los componentes. - [ variable ] : Para incluir información a componentes. - \*ngFor o \*ngIf que nos servirán como if y for, los veremos en profundidad a continuación.

Directivas estructurales

Anteriormente hemos visto algunos tipos de directivas, ahora veremos en profundidad algunas de las directivas estructurales más importantes.

**NgIf**

\*ngIf=”conditionVar”

Nos permite decidir sobre la existencia de elementos y componentes. Funciona igual que la instrucción if en typescript y espera una variable tipo boolean que será evaluado.

*Ojo: Si esta en negativo el elemento en el dom no existe, por lo cual los componentes anclados bajo este elemento, se destruirán o crearán y afectará a su ciclo de vida.*

**NgFor**

Lo usaremos normalmente para generar contenido de forma dinámica.

\*ngFor=”let correo of listaCorreos”

Recorremos la lista de correos como cualquier función iteradora, acepta elementos iterables.

\*ngFor=”let correo of listaCorreos; index as i;”

Recorremos y además podemos tener referencia de la cuenta del for.

Ejemplo:

<div \*ngFor="let correo of listaCorreos; index as i;"> <p> {{ correo.titulo }} <span>Número {{ i }}</span></p> <p> {{ correo.cuerpo }} </p> </div>

En el ejemplo podemos ver que se ha usado en un elemento div y que dentro de este elemento, se pueden usar las variables para referenciar al contenido que estamos recorriendo.

**Más Documentación**

* **[ngIf](https://angular.io/api/common/NgIf" \t "_blank)**
* **[ngFor](https://angular.io/api/common/NgForOf" \t "_blank)**
* **[Otras](https://angular.io/guide/structural-directives" \t "_blank)**

Formularios Rectivos

<https://angular.io/guide/reactive-forms>

Cada vez que usamos la directiva de un componente para crearlo dentro de otro componente, estamos creando un nuevo componente hijo de nuestro componente actual.

Rápidamente veremos la necesidad de poder comunicarnos con este nuevo componente después de haber sido creado.

Las comunicaciones que vamos a ver, tendrán el objetivo de transmitir una información desde el componente pare al hijo o disparar una funcionalidad del componente hijo. Para ello podemos usar solo typescript o también usar el “template”.

## Preparación del componente hijo

El componente hijo con el que nos vamos a comunicar, debe disponer de por ejemplo una variable en la que vamos a cambiar su valor. También puede tener funcionalidades que podamos acceder desde el componente padre.

Como parte extra cuando queremos que una variable esté accesible desde el “template”, debemos acompañar las variables de un decorador de angular @Input.

*Un decorador es una herramienta que nos proporciona angular para indicar una configuración extra que se le aplica a algo de forma dinámica. Cómo lo hacemos al definir un componente con @Component.*

@Input() variable: tipo;

## Comunicación

Según nuestras necesidades usaremos una forma u otra de comunicarnos con el componente, a continuación veremos algunas de ellas.

### En el “template”

Veremos en el siguiente ejemplo como se le indica un valor para el título a un componente que hijo que estamos creando desde el padre.

<componente-hijo titulo=”Mi Titulo”></componente-hijo>

Ahora en lugar de un valor, le indicaremos lo mismo pero usando una variable.

<componente-hijo [titulo]=”variableTitulo”></componente-hijo>

### Mediante typescript

A veces necesitaremos realizar este mismo proceso usando typescript, y además en un momento posterior al inicio de la aplicación.

Por ejemplo, cuando dejamos que un usuario introduzca un dato y tras ello, queremos actualizar un valor o incluso activar una función del hijo.  
Para ello, vamos a usar otro decorador @ViewChild, pero esta vez en el componente padre para poder tener acceso al componente hijo.

@ViewChild(componente-hijo) variableComponenteHijo: HijoComponent;

Ya podemos acceder a sus variables y métodos, por ejemplo usando una función en nuestro componente padre.

enviarTituloAlHijo() { this.variableComponenteHijo.titulo = “Mi Titulo” }

Cuando se ejecute la función enviarTituloAlHijo, el valor de la variable del hijo “título”, cambiará.

Podríamos ejecutar un método del componente hijo también.

enviarTituloAlHijo() { this.variableComponenteHijo.functionDelHijo(); }

## Más Documentación

* [**@Input**](https://angular.io/api/core/Input)
* [**@ViewChild**](https://angular.io/api/core/ViewChild)
* [**@Component**](https://angular.io/api/core/Component)

Para llevar a cabo la comunicación entre un componente hijo y su padre, podemos recurrir a varios métodos, para facilitar la compresión e ir poco a poco, vamos a ver sólo el uso de eventos. En el próximo tema veremos otras herramientas que podremos aplicar también en estos casos.

## Eventos

Los eventos, son una herramienta que nos permite “avisar” a otros componentes en el momento que queramos. Por ejemplo, un evento podría ser un click de un botón o el fin de la carga de un servicio. Cuando se “dispara” este evento, todos los componentes que estuvieran a la escucha, son avisados y se les traslada información.

Con un simple decorador de angular, podemos crear nuestros propios eventos en nuestros componentes y permitir a nuestro componente padre ser avisado cuando lo decidamos.

## Uso de eventos en el hijo

En nuestro componente hijo, vamos a usar el decorador de angular @Output().  
Este decorador estará asociado a una variable de tipo evento, que crearemos en nuestro componente hijo y que dispondrá de un método para disparar el evento.

Vamos a construir un ejemplo simple con un evento llamado “hijoAvisando”. Para ello hacemos:

@Output() hijoAvisando: EventEmitter<any> = new EventEmitter();

Ahora la variable hijoAvisando, dispone del método emit(value?: T). Con este método podemos “disparar” el evento e incluir alguna información.

En nuestro componente hijo:

let mensaje = “hola”; this.hijoAvisando.emit(mensaje)

## Uso de eventos en el padre

Ya hemos configurado nuestro componente hijo para emitir un evento con un mensaje, ahora nos toca hacer lo propio en el componente padre.

Para ello vamos a crear primero en typescript una función para asignarla al evento que va a emitir nuestro componente hijo. Llamaremos a esta función “mensajeRecibido” y se le parará un mensaje de tipo texto que escribirá en el log.

mensajeRecibido(mensaje: string) { console.log(mensaje); }

Ahora solo nos queda incluir en nuestro “template” del padre, una referencia para que el evento del hijo, “dispare” la función que acabamos de crear en el padre.

<componente-hijo (hijoAvisando)=”mensajeRecibido($event)”> </componente-hijo>

## Más Documentación

* **[@Output](https://angular.io/api/core/Output" \t "_blank)**
* **[EventEmitter](https://angular.io/api/core/EventEmitter" \t "_blank)**

Es un tipo de dato que actúa como un canal de comunicación y forma parte de las librería Rxjs, el concepto de observable es como el de evento, nos sirve para esperar “avisos” con información cuando “escuchamos”. En este caso para “escuchar” “avisos” nos **Suscribimos** a un **Observable**.

Lo primero a tener en cuenta cuando trabajamos con typescript (o javascript en general) es que es un idioma asíncrono, al igual que lo son la mayoría de acciones en las que debemos obtener información o esperar la carga de elementos.

Los **Observables** nos ayudan en la tarea de sincronizar todas esas acciones que no se producen de inmediato. Para ello creamos variables de este tipo para emitir avisos de cambios y/o información.

## Definir un observable

Los observables son tipos de datos devueltos muy comúnmente por muchas funciones de angular o typescript. Es común usar en el nombre de estos, el símbolo $ para dejar claro que tipo de dato se trata.

myObservable$ = Observable<any[]>;

Podemos construir un observable fácilmente con un array de números:

myObservable$ = of(1, 2, 3);

## Suscripciones

Como se ha visto, los observables son un tipo de dato, una funcionalidad de estos tipos es la de suscribirse indicando qué queremos hacer cuando el observable “cambie”.

Para ello debemos indicarle que hacer mediante funciones, al suscribirnos nos permite indicar 3 funciones.

La primera es obligatoria y será la que se ejecute cuando el observable emita un “cambio”.

Las siguientes funciones son para indicar qué hacer en caso de error y la última es la que se ejecuta cuando se finaliza el observable. Veamos un ejemplo completo.

myObservable$.subscribe( (datos) => { // acción al recibir datos }, (error) => { // acción solo para el caso de error }, () => { // acción al finalizar } );

## Pipes

Esta utilidad nos permite aplicar cambios a los valores recibidos mediante el observable, antes de que se entregue su valor final a los suscriptores. Vamos a ver sólo cómo usar una función para editar los valores usado map.

myObservable$.pipe( map( data => { *// cambios a los datos* } ) );

## Terminar suscripción

Más adelante veremos el ciclo de vida de los componentes de angular, pero una cosa que importante saber en este punto, es que cuando nos suscribimos a un observable, siempre debemos cerrar la suscripción al terminar, de lo contrario se mantendrá activa y generará errores.

Para ello solo debemos usar lo siguiente:

const mySubscription = myObservable$.subscribe(); if(!mySubscription.closed){ mySubscription.unsubscribe(); }

## Más Documentación

* **[Observables](https://angular.io/guide/observables" \t "_blank)**
* **[Pipe](https://angular.io/guide/pipes" \t "_blank)**
* **[Pipe como operador](https://angular.io/guide/rx-library" \l "operators" \t "_blank)**

Podemos entender un servicio como un componente sin un “template asociado”. Es otra nueva forma de organizar nuestro código, aparte de los componentes

Aquí encontraremos un poco de confusión, esto es debido a que los servicios son usados para varias cosas.

* Los servicios “proveedores”, serán usados para obtener información. Normalmente de servicios web, pero también podrían ser desde bases de datos, ficheros, etc…
* Los servicios para “compartir”, estos se usan para compartir una información entre varios componentes.
* Los servicios “inteligentes”, estos los usaremos para contener la lógica de nuestra aplicación. Nos ayudarán a reutilizar más nuestro código.

## CLI Angular

Podemos usar CLI de Angular para generar un nuevo servicio con el siguiente comando:

ng generate service <name> [options]

Simplificado en:

ng g s <name> [options]

## Más Documentación

* **[CLI service](https://angular.io/cli/generate" \l "service" \t "_blank)**

SIGUIENTE

1. Acceder con una cuenta de google **[aquí](https://console.developers.google.com/" \t "_blank)**
2. Crear un nuevo proyecto.
3. Habilitar APIs y servicios
4. Buscar en la lista “Gmail Api”, escogerla y habilitarla.

## Configurar OAuth

Para conectarnos con google es necesario configurar este paso, para ello:

1. Vamos al menú pantalla de consentimiento de OAuth.
2. Ponemos un nombre al proyecto, yo use IniciacionAngular8
3. En permisos para acceder a las API de Google habilitamos:  
   email  
   profile  
   openid  
   ../auth/gmail.labels  
   ../auth/gmail.readonly  
   ../auth/gmail.send
4. Guardamos

## Configurar credenciales

Para poder usar en nuestro proyecto las apis de google, necesitamos configurar nuestra aplicación y obtener los datos necesarios para usar la api.

1. Indicamos proyecto web
2. Ponemos un nombre, yo use IniciacionAngular8
3. Orígenes ponemos: http://localhost:4200 (Más rutas si fuera necesario)
4. Redirección ponemos alguna si la vamos a usar.
5. Crear

Ya tenemos el ID que usaremos en los siguientes pasos.

395684135333-7cs6mldgfgr2q531bu51el0j4hdosv0s.apps.googleusercontent.com

*Siempre que volvamos a credenciales, podremos consultar los tokens de nuestros proyectos.*

En esta práctica vamos a crear un nuevo servicio y lo vamos a concectar con la lista de correos para recibir los correos de la bandeja de entrada de Gmail.

Para esta tarea, vamos a ayudarnos de una libreria npm **[más info](https://www.npmjs.com/package/ng-gapi" \t "_blank)**

npm i --save ng-gapi

En esta práctica vamos a usar la session del navegador para guardar el token del usuario, tras logearse en la aplicación.  
Para ello usaremos dos funciones sencillas:

Para guardar (objeto):

sessionStorage.setItem('clave', JSON.stringify(objeto));

Para recuperar, transformando en objeto el resultado:

JSON.parse(sessionStorage.getItem('clave'));

Para eliminar el contenido:

sessionStorage.removeItem('clave');

## Git

**[Enlace GitHub](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/tree/clase4-9-OAuthGoogle" \t "_blank)**

git checkout clase4-9-OAuthGoogle

* Rama: **[clase4-9-OAuthGoogle](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/compare/clase4-7-servicio-interconexion...clase4-9-OAuthGoogle" \t "_blank)**

## Más Documentación

* **[Auth0 y Gmail](https://developers.google.com/gmail/api/auth/about-auth" \t "_blank)**
* **[Permisos para Gmail](https://developers.google.com/gmail/api/auth/scopes?utm_campaign=gsuite_series_gmailauthscopes_120716&utm_source=gdev&utm_medium=yt-desc" \t "_blank)**
* **[API Gmail](https://developers.google.com/gmail/api/v1/reference/" \t "_blank)**

Angular crea aplicaciones de una sola página. Esto quiere decir que una vez que se carga nuestra web, ya no vuelve a recargar la página.

Las funcionalidad de rutas, realmente elimina componentes o los crean en la página. Lo que permite que el rendimiento sea bueno.

## CLI Angular

La forma más rápida de incluir las rutas en un proyecto, es indicarlo al crear el proyecto con CLI de Angular, ya que es la primera pregunta que nos lanza.

Pero si tras generar el proyecto se quieren crear, es tan fácil como usar CLI con el siguiente comando para generar el fichero de configuración de rutas.

ng generate module rutas --routing --flat --module=app

Simplificado en:

*ng* *g* *m* *rutas* --*routing* --*flat* -*m=app*

Este comando crea un módulo llamado rutas, que incluirá routing **(—routing)**, se creará en la carpeta raíz **(—flat)** y se incluirá en el appModule **(—module=app)**, de forma automática.

Este comando nos va a generar 2 ficheros, uno de ellos es idéntico al generado al iniciar la aplicación (rutas-routing.module.ts), pero en este caso al ser una vez iniciada, no crea además otro fichero de módulo(rutas.module.ts).

Este último fichero, nos ayudará a mantener más limpios y organizados nuestros ficheros de ruta, sobre todo cuando generamos varios de ellos. En los ejemplos que vamos a ver no vamos a necesitar ese fichero, por lo cual podemos eliminarlo.

## Fichero routing

Vamos a ver cómo debemos configurar las rutas de nuestra aplicación, para ello debemos prestar atención a la variable del fichero llamada routes de tipo Routes. Esta variable está formada por un array de objetos que indican cómo debe funcionar las rutas.

const routes: Routes = [];

Los objetos en typescript se escriben usando las {}; Para saber que nombres y configuraciones podemos usar, se definen las interfaces que veremos en profundidad más adelante.

En nuestro caso práctico, vamos a ver la más usada.

{ path: string, component: Component }

Incluyendo en path un texto indicando la ruta, además se puede indicar parámetros que se esperan recibir con la ruta y más cosas. Podemos ver en la documentación completa todos las opciones.

En component se indica el componente que se cargará, para facilitar esta parte, nuestro proyecto tiene una carpeta dedicada exclusivamente (/views) para guardar los componentes que forman parte del menú, donde a partir de ellos se crea cada página.

Como ejemplo usaremos:

{ path: '', component: HomeComponent } (Carga Home en la ruta /) { path: 'home', component: HomeComponent } (Carga Home en la ruta /home) { path: 'mail', component: MailComponent } (Carga el Mail en la ruta /mail)

## Parámetros en la ruta

Usando la anterior configuración, hemos adelantado que podíamos recibir parámetros por medio de las rutas, para ellos vamos a verlo con más detenimiento. En este fichero es necesario indicar que se envía, sinó no se podrá recuperar de la misma forma.

Usando por ejemplo:

{ path: 'mail/:id', component: MailComponent }

Ya podríamos obtener en el componente Mail, el valor de “id”, enviado de esta manera en la ruta:

/mail/32 (Se estaría enviando el número 32)

## Usar las rutas en nuestra aplicación

Para poner en marcha la carga de rutas en nuestra aplicación, vamos a usar un componente de angular que se encarga de hacer todo el trabajo. Solo tenemos que elegir el lugar donde queremos realizar la sustitución y dependerá de nuestros gustos.

Normalmente, se suele indicar en el fichero principal “app.component.html” el componente menú (app-menu) y la carga de rutas (router-outlet). Con ello ya está listo, sería así:

<app-menu></app-menu> <router-outlet></router-outlet>

## Rutas en “template” para menú

Veremos a continuación como poder indicar en los “templates” que queremos navegar a otro componente de la aplicación.

Esta parte, normalmente está, al menos en el menú, por lo tanto vamos a ver un ejemplo con esta utilidad. En el “template” de un componente escribimos lo siguiente:

<a routerLink="/home"> Home </a>

Estamos indicando que al pulsar el enlace “Home” se dirigirá al fichero de configuración que escribimos anteriormente, en él buscará la ruta /home entre los path, y a continuación cargará el componente indicado.

Si además de enviar la ruta, queremos enviar algún parámetro como en el caso anterior de /mail/32, seria de la siguiente manera. Previamente el valor está almacenado en una variable (identificadorMail) del componente, así que solo tenemos que poner:

<a [routerLink]="['/mail', identificadorMail]"> Home </a>

## Rutas en componentes

Solo nos queda ver como podemos acceder a las rutas desde los componentes. Siguiendo con los ejemplos de /mail/id, vamos a ver cómo obtener el valor enviado.

En el constructor del componente vamos a incluir la declaración de route para poder recoger la información que queramos de la ruta. (Incluso las ruta anterior y más información)

constructor( private route: ActivatedRoute, private router: Router, ) *{}*

Ya podemos usar route para recoger el id, así:

let id = this.route.snapshot.paramMap.get('id');

Otra utilidad es por ejemplo, navegar a otra pantalla desde typescript, usando una función. para ello es tan fácil como:

irAHome() { this.router.navigate(['/home']); }

## Más Documentación

* **[Rutas](https://angular.io/guide/router" \t "_blank)**
* **[CLI crear rutas](https://angular.io/cli/generate" \l "module" \t "_blank)**
* **[Lo básico](https://angular.io/guide/router" \l "the-basics" \t "_blank)**
* **[Fichero de routing](https://angular.io/guide/router" \l "configuration" \t "_blank)**
* **[Router-outlet](https://angular.io/guide/router" \l "router-outlet" \t "_blank)**
* **[Router-links](https://angular.io/guide/router" \l "router-links" \t "_blank)**
* **[Ejemplo aplicación](https://angular.io/guide/router" \l "the-sample-application" \t "_blank)**

En esta práctica vamos a empezar a usar las rutas, el menú de navegación y las vistas. Para ello vamos a crearnos las vistas necesarias, configurar el archivo de rutas de app-routing y vamos a crear un componente menú para poder cambiar entre vistas.

## Git

**[Enlace GitHub](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/tree/clase5-2-navegacion" \t "_blank)**

git checkout clase5-2-navegacion

* Rama: **[clase5-2-navegacion](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/compare/clase4-10-ServiciosHttp...clase5-2-navegacion" \t "_blank)**

SIGUIENTE

“Material” es una librería de componentes listos para usar, nos permiten alcanzar grandes resultados con mínimo desarrollo.

Vamos a ver la **[Web](https://material.angular.io/components/categories" \t "_blank)** donde se muestran todos los componentes que podemos usar, las intrucciones y ejemplos de uso.

Para el estilo de nuestra aplicación vamos a usar estos componentes y solo usaremos bootstrap para la colocación de elementos (“responsive”).

En esta práctica vamos a sustituir el html básico que hemos construido durante el curso, por los componentes de “material”.

Para incluir material en angular existen varias vias, pero la más fácil es usar CLI con el siguiente comando:

ng add @angular/material

Al ejecutar el comando comenzará a instalar lo necesario (En lugar de manualmente). CLI nos realizará preguntas respecto de que instalar así que contestamos:

1. El primero(Tema). Tema queremos incluir por defecto, podemos incluir cualquiera.
2. Si(HammerJS). Algunos componentes usan la librería HammerJS y nos pregunta si la incluye al proyecto.
3. Si(Animaciones). Este módulo no vamos a verlo en este curso, pero puede que algún componente lo necesite para funcionar con normalidad.

Ya CLI ha instalado las librerías npm necesarias y ha importado en los archivos pertinentes las partes que necesita. Podemos ver el resumen:



NPM

UPDATE package.json (1413 bytes) added 3 packages from 4 contributors in 7.214s



Importaciones y configuraciones.

UPDATE src/main.ts (391 bytes) UPDATE src/app/app.module.ts (2548 bytes) UPDATE angular.json (3818 bytes) UPDATE src/index.html (504 bytes) UPDATE src/styles.scss (181 bytes)

Tambien vamos a importar bootstrap en nuestro index.html para usarlo para posicionar los elementos.

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

## Git

**[Enlace GitHub](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/tree/clase6-2-material" \t "_blank)**

git checkout clase6-2-material

* Rama: **[clase6-2-material](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/compare/clase5-2-navegacion...clase6-2-material" \t "_blank)**

## Más Documentación

* **[Incluir “material”](https://material.angular.io/guide/getting-started" \t "_blank)**
* **[HammerJS](http://hammerjs.github.io/" \t "_blank)**
* **[Componentes “Material”](https://material.angular.io/components/categories" \t "_blank)**
* **[Incluir Bootstrap](https://getbootstrap.com/docs/4.3/getting-started/introduction/" \t "_blank)**
* **[Utilidades Bootstrap](https://getbootstrap.com/docs/4.3/getting-started/introduction/" \t "_blank)**

Son la forma de controlar los colores de estilos generales y particulares. Además de ser la solución para permitir cambiar esta parte de forma dinámica y muy fácilmente.

Los “Themes” disponen de paletas de colores y de dos tipos de estilos para usarlos.  
Podemos optar por usar varios temas ya configurados, directamente como css para incluir en la aplicación o podemos crear los nuestros propios.

## Opciones de colores

En los temas existen 3 colores básicos y se definen por los siguientes nombres.

* primary
* accent
* warn

Al usar “Material” y “Themes” juntos, podemos definir que color debe tomar un componente, indicando en el “template” donde lo definimos los valores anteriores a la variable “color”:

<mat-progress-spinner color="warn"></mat-progress-spinner>

## Creando nuestros temas

Para crear nuestros temas, debemos crear un fichero css donde vamos a tener que definir unos cuantos parámetros. Lo crearemos en la ruta (src/Themes) y se llamará “myTema.scss”. Si la creamos dentro de src/app/ nos fallará al iniciar angular.

Para ello importamos y procedemos a crear el tema:

@import '~@angular/material/theming'; @include mat-core();

### Paletas

Debemos seleccionar 3 paletas de colores distintas para formar un tema, cada uno de esos colores se usará para **“primary”**, **“accent”** y **“warn”**.

Seleccionamos en 3 variables, de la siguiente manera, las paletas seleccionadas. Podemos ver todas las paletas disponibles en los enlaces finales

$palette-primary: mat-palette($mat-grey, 500); $palette-accent: mat-palette($mat-light-blue); $palette-warn: mat-palette($mat-green, 800);

La función mat-palette permite escoger la paleta y además su color base

### Estilos

Una vez seleccionadas las paletas de colores, debemos escoger qué estilo utilizar.  
Tenemos mat-light-theme (colores claros contra oscuros) o mat-dark-theme (colores oscuros contra claros)

Usamos el primero por ejemplo:

$light-theme: mat-light-theme( $palette-primary, $palette-accent, $palette-warn );

## Usar nuestros temas

Una vez realizados todos los pasos anteriores, pasamos ahora a importar en nuestro fichero general de estilos de angular, el tema creado.

Para ello debemos incluir una importación en el fichero de estilos generales del proyecto de angular que se encuentra en src/styles.scss

@import ‘Themes/myTema.scss'; @include angular-material-theme($light-theme);

## Varios temas a la vez

Podemos repetir el proceso anterior y crear cuantos temas queramos, una vez tenemos varios, debemos incluir un tema por defecto en nuestra aplicación (como se ha hecho anteriormente) y tenemos la posibilidad de crear clases css, con esos otros temas, para aplicar a los elementos.

Para ello incluimos en nuestro fichero styles.scss clases de la siguiente manera:

.other-theme { @include angular-material-theme($other-theme); }

## Más Documentación

* **[“Themes”](https://material.angular.io/guide/theming" \t "_blank)**
* **[Paletas](https://material.io/archive/guidelines/style/color.html" \l "color-color-palette" \t "_blank)**

En esta práctica vamos a construir una interfaz para los correos en nuestra aplicación, así la podremos usar como tipo y nos ayudara en el desarrollo.

*Una interfaz no es mas que la declaración de un objeto y sus propiedades.*

Nos ayudaremos de CLI, para como hasta ahora, para crearla más rápidamente.

ng g interface correo

## Git

**[Enlace GitHub](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/tree/clase7-1-Interfaces" \t "_blank)**

git checkout clase7-1-Interfaces

* Rama: **[clase7-1-Interfaces](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/compare/clase6-4-Themes...clase7-1-Interfaces" \t "_blank)**

## Más Documentación

* **[Comando generate interface CLI](https://angular.io/cli/generate" \l "interface" \t "_blank)**
* **[Más información sobre interfaces](https://www.typescriptlang.org/docs/handbook/interfaces.html" \t "_blank)**

En esta práctica vamos a usar las pipes para simplificar el código que usamos en las llamadas para recoger los mensajes, de manera que pasen por una “pipe” (tubería) y cuando lleguen al componente ya tengan forma de “correo”. Este uso de pipe pertenece a la parte operadores de observables de Angular.

observableRespuesta.pipe(map( data => data ))*;*

Además las pipes también se usan en los templates para convertir datos de una forma a otra, en este curso haremos un ejemplo usando una pipe para poner en mayúsculas el nombre del usuario en la pantalla principal.

{{ value | uppercase }}

## Git

**[Enlace GitHub](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/tree/clase7-2-Pipes" \t "_blank)**

git checkout clase7-2-Pipes

* Rama: **[clase7-2-Pipes](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/compare/clase7-1-Interfaces...clase7-2-Pipes" \t "_blank)**

## Más Documentación

* **[Pipe como operador](https://angular.io/guide/rx-library" \l "operators" \t "_blank)**
* **[Pipes en templates](https://angular.io/api/core/Pipe" \t "_blank)**
* **[Pipe para minúsculas](https://angular.io/api/common/LowerCasePipe" \t "_blank)**
* **[Pipe para mayúsculas](https://angular.io/api/common/UpperCasePipe" \t "_blank)**
* **[Comando generate pipe CLI](https://angular.io/cli/generate" \l "pipe" \t "_blank)**

La inyección de dependencias se dá en angular, principalmente en los servicios, esto es debido a que son los principales elementos que se comparten por la aplicación de manera transversal.

Podemos ver como ejemplo el servicio de avisos, este servicio que hemos desarrollado durante el curso se comparte en toda la aplicación y cualquier componente puede importarlo y usarlo.

Lo importante de aquí, es que todos los componentes comparten el mismo servicio, no copias del mismo servicio, por ello se comparte y persiste la información contenida en ellos.

## Decorador @Injectable

Usamos este decorador en los servicios para compartirlos con el resto de la aplicación sin necesidad de incluirlos en app.module. (Dependiendo de que necesitemos)

Esta utilidad fue incluida en versiones avanzadas de angular y en Angular 8 están muy consolidadas.

En el día a día con Angular 8 prácticamente siempre usaremos la misma configuración. Que es la que nos genera CLI al generar los servicios de manera normal.

@Injectable({ providedIn: 'root' })

Con esto estamos posibilitando que cualquier componente pueda requerirlo en su constructor sin tener que especificar nada mas, así:

constructor(public servicioAvisos: AvisosService) *{ }*

*Aprovecharé para explicar que no es lo mismo usar private/public delante de la variable que importamos en el constructor, que no poder nada. Lo anterior es equivalente a lo siguiente:*

servicioAvisos: AvisosService; constructor(servicioAvisos: AvisosService) *{ this.servicioAvisos = servicioAvisos; }*

*Ya vemos que nos han simplificado un poco el código de esta manera, atentos a estos detalles ya que serán los causantes de más de un error.*

## Jerarquía

No entraremos mucho en detalle, solo vamos a ver que angular va a “resolver” nuestra petición de una dependencia, de manera jerárquica.

1. Root
2. NgModules (app.module)

*Por lo que al registrar nuestros servicios con la forma habitual de CLI, estarán disponibles en toda la aplicación.*

## Más Documentación

* **[@Injectable](https://angular.io/api/core/Injectable" \t "_blank)**
* **[Jerarquía de inyección](https://angular.io/guide/hierarchical-dependency-injection" \t "_blank)**
* **[Más sobre inyección de dependencias](https://angular.io/guide/dependency-injection-providers" \l "dependency-providers" \t "_blank)**
* **[Más sobre inyección de dependencias en Angular](https://angular.io/guide/dependency-injection" \t "_blank)**

En esta práctica vamos a construir un módulo para guardar todos los componentes de “material” que hemos ido incluyendo.

*Los módulos son “paquetes” de componentes, servicios, etc …*

También hemos visto con OAuth, que se pueden configurar al incluirse en app.module.

Para crear un nuevo módulo usamos CLI:

ng g module material

Configuraremos el módulo para exportar los módulos de angular y limpiar un poco el app.module.

## Git

**[Enlace GitHub](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/tree/clase7-4-Modulacion" \t "_blank)**

git checkout clase7-4-Modulacion

* Rama: **[clase7-4-Modulacion](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/compare/clase7-2-Pipes...clase7-4-Modulacion" \t "_blank)**

## Más Documentación

* **[Módulos](https://angular.io/api/core/NgModule" \t "_blank)**
* **[Comando generate module CLI](https://angular.io/cli/generate" \l "module" \t "_blank)**

El ciclo de vida de un elemento de angular son las fases por las que pasa un elemento desde que se crea hasta que se destruye.

Determinadas acciones no estarán disponibles en algunos momentos del ciclo de vida de un componente. Por ejemplo, si usamos un pasamos información de un componente padre a un componente hijo, no podremos recuperar esa información en el constructor. Necesitaremos hacerlo después de que el componente se inicie.

Angular nos provee de unas funciones que se activan cuando se pasa por ese momento del ciclo de vida, para poder realizar las acciones que necesitemos de manera fácil.

*Un ejemplo claro de ciclo de vida ocurre al usar \*ngIf con componentes, estos se crean al ser verdadera su condición, pero se destruyen cuando es falsa.*

## Ciclo de vida simplificado

1. “constructor()”

* Todo empieza con él, se ejecuta lo primero.

1. “ngOnInit()”

* Se ejecuta tras recibir las propiedades de entrada, aquí solemos inicializar todo.

1. “ngAfterContentInit()”

* Se ejecuta después de que Angular proyecte contenido externo en la vista del componente (ng-content).

1. “ngAfterContentChecked()”

* Se ejecuta tras comprobar el contenido proyectado en el paso anterior.

1. “ngAfterViewInit()”

* Se ejecuta tras iniciar todas las vistas y subvistas del elemento.

1. “ngOnDestroy()”

* Este es el último en ejecutarse, justo antes de destruir el elemento. Aquí hay que tener especial atención porque debemos destruir todas las subscripciones a observables que se iniciaran en el elemento, de lo contrario, seguirán ejecutándose.

*Si usamos una función setInterval y no eliminamos la subscripción al destruir el componente, se seguirá ejecutando.*

## Ciclo de vida cíclico

Aparte del ciclo anterior, existe otro ciclo que se repite indefinidamente, esto permite a angular detectar los cambios y realizar acciones.

1. “ngOnChanges()”

* Continuamente revisa los cambios realizados en las variables.

1. “ngDoCheck()”

* Después de cada ngOnChanges, nos permite ejecutar comprobaciones manuales.

1. “ngAfterContentChecked()”

* Esta la vimos en el ciclo anterior, también se repite detrás de ngDoCheck en cada ciclo.

1. “ngAfterViewChecked()”

* Se ejecuta en cada ciclo después del anterior y tras comprobar las vistas y subvistas del elemento.

## Destruir la subscripción a observables

Vamos a ver como seria el ciclo de vida de un observable en un componente y como se debe destruir.

Declaramos la subscripción al inicio:

recibidosSubscription: Subscription;

Momento de suscribirse al observable:

this.recibidosSubscription = this.gmail.getRecibidos().subscribe(...);

Destrucción de la subscripción, debemos comprobar si ya fue eliminada para que no produzca errores.

ngOnDestroy(){ if(!this.recibidosSubscription.closed){ this.recibidosSubscription.unsubscribe(); } }

## Git

**[Enlace GitHub](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/tree/clase7-5-CicloVida" \t "_blank)**

git checkout clase7-5-CicloVida

* Rama: **[clase7-5-CicloVida](https://github.com/carlosgarciaopenwebinar/iniciacionAngular8/compare/clase7-4-Modulacion...clase7-5-CicloVida" \t "_blank)**

## Más Documentación

* **[Cliclo de vida](https://angular.io/guide/lifecycle-hooks" \t "_blank)**
* **[Observables](https://angular.io/guide/observables" \l "basic-usage-and-terms" \t "_blank)**