



ANGULAR

CONCEPTOS AVANZADOS

DIEGO LÓPEZ GARCÍA

@CodingPotions



Angular, conceptos avanzados

CodingPotions

March 23, 2019

Contents

Introducción	4
Sobre el autor	4
Debug de aplicaciones Angular	6
ng.probe(\$0).componentInstance	6
ng.profiler.timeChangeDetection()	7
Augury	7
debugger	9
Usos de ng-template, ng-container y ngTemplateOutlet	10
ng-template	10
ng-container	11
ngTemplateOutlet	12
Cómo usar SASS con Angular	13
Decoradores Angular	14
Decoradores de clase	14
Decoradores de atributos y parámetros	15
Decoradores de métodos	16
Cómo crear tus propios decoradores	16
Gestión de multiples entornos	18
Creación de más entornos	20
Aislamiento de estilos para los componentes	22
:host	22
::ng-deep	23

Ciclo de vida de los componentes	24
ngOnInit	24
ngOnChanges	25
AfterViewInit	26
OnDestroy	27
Directivas	28
Directivas de atributo	28
Directivas estructurales	30
Observables y promesas	33
Animaciones	36
PWA	42
SEO en Angular	46
Cómo estructurar un proyecto grande con Angular	50
La carpeta core	50
La carpeta shared	51
La carpeta pages	51
La carpeta components	51
Otras consideraciones	52
Lazy components	53
Testing unitario	55
Tests unitarios con Jasmine	56
Tests unitarios con Angular	58
Testeando clases en Angular	59
Usando el servicio real	60
Creando un servicio virtual (mockeando)	61
Mediante del uso de spy de Jasmine	62
Testeando llamadas asíncronas	63
Accediendo a la vista	64
Testing de llamadas http	65
Algunos consejos para terminar	66
Conclusiones	68

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

— Linus Torvalds

Introducción

¡Hola! Lo primero, muchas gracias por adquirir este libro. Me hace mucha ilusión que confíes en mi para seguir aprendiendo y formándote sobre Angular.

Lo que vas a leer a continuación es un libro sobre conceptos avanzados de Angular, es decir, con este libro presupongo que ya tienes una base de Angular, que al menos sabes crear componentes, rutas y servicios.

¡ATENCIÓN! Este libro esta pensado para ser usado a partir de la versión 2 de Angular, si tu intención es aprender cosas de AngularJS (primera versión), lo que hay escrito no te va a servir.

Los conceptos de este libro no los he ordenado siguiendo ningún orden en particular, por lo que los puedes leer en el orden que quieras.

Si te gusta el libro y quieres apoyarme puedes mandar un tweet del estilo:

Acabo de conseguir el ebook de #angular sobre conceptos avanzados
de @CodingPotions aquí: <https://leanpub.com/angular-avanzado>

También me gustaría pedir perdón porque al ser un libro en PDF no vas a poder copiar y pegar el código que aparecé aquí escrito. Si necesitas que te pase algún código de los que aquí aparecen puedes contactar conmigo por Twitter: @CodingPotions

Sobre el autor

Antes de empezar me gustaría presentarme, soy Diego López, desarrollador frontend desde hace varios años y dueño y escritor del blog de **CodingPotions**

<https://codingpotions.com>.



Puedes contactar conmigo mediante mi email: **diego.lopezgarcia@hotmail.com** o mandando un mensaje a mi cuenta de Twitter **[@Codingpotions](https://twitter.com/Codingpotions)**

Debug de aplicaciones Angular

Cuando tienes un error en la aplicación o algo no ha salido como esperabas, puede resultar muy útil hacer debug, es decir, ver en tiempo de ejecución el valor de variables y elementos en tiempo real.

Uno no puede programar algo sin tener un solo fallo, de no ser que estés usando una estrategia TDD (Test Driven Development) que sea infalible por eso es fundamental conocer lo que ofrece Angular para hacer debug.

ng.probe(\$0).componentInstance

Con este comando vas a poder el estado de un componente desde la consola del navegador. Para usarlo abre el inspector de elementos de tu navegador y selecciona un componente en la pestaña de **Elements**. A continuación ejecuta:

```
ng.probe($0).componentInstance
```

En este caso \$0 es una variable global que se traduce al último elemento seleccionado en el inspector.

Es importante saber que este comando solo servirá si tenemos activado el modo debug de Angular. Normalmente el modo debug está desactivado en producción cuando haces build, para que lo tengas en cuenta.

The screenshot displays the Chrome DevTools interface. The **Elements** panel on the left shows the DOM tree with the `tm-shell-main-nav` component selected. The **Styles** panel on the right shows the default styles for the `main-nav` component, including `display: block`, `background-color: #FFF`, and `height: 56px`. The **Console** panel at the bottom shows the output of the `ng.probe($0).componentInstance` command, displaying the `MainNavComponent` instance with various properties and methods.

ng.profiler.timeChangeDetection()

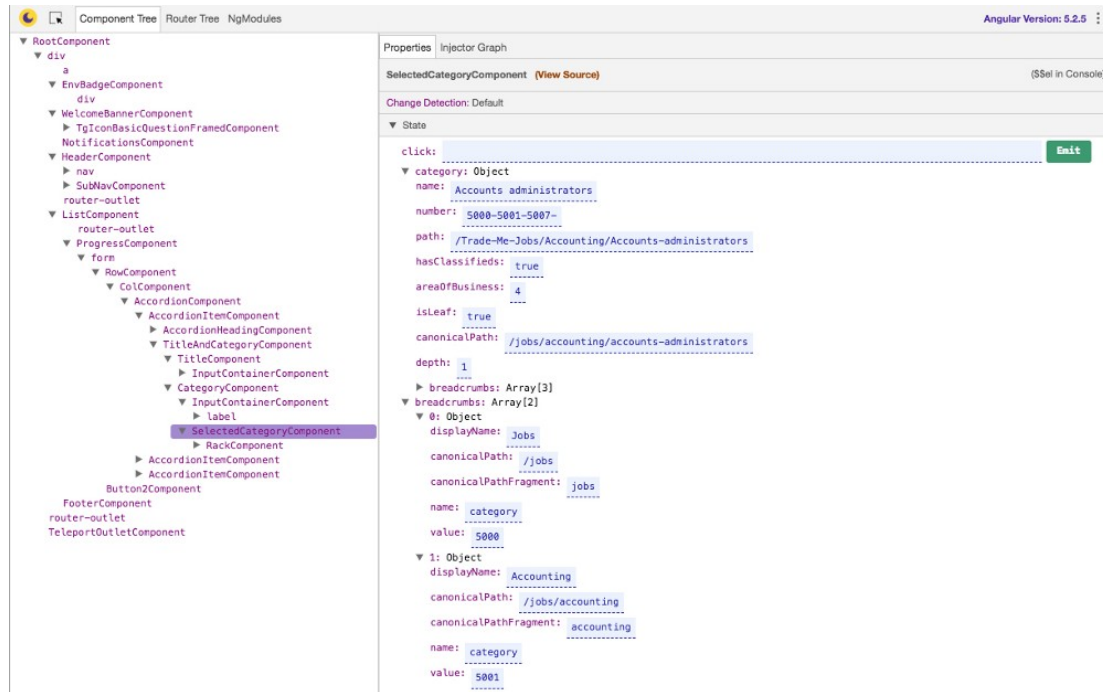
El comando `profile` sirve para medir cosas, por el momento Angular solo ofrece medir el **changeDetection**. `changeDetection` mide el tiempo que pasa cada vez que Angular detecta un cambio y actualiza la vista. Esto nos puede dar una medida de rendimiento, si tarda más de 3ms en cada cambio nos tenemos que plantear mejorar su rendimiento.

```
> ng.profiler.timeChangeDetection()
ran 77538 change detection cycles
0.01 ms per check
< ▶ ChangeDetectionPerfRecord {msPerTick: 0.0064484510820500916, numTicks: 77538}
> |
```

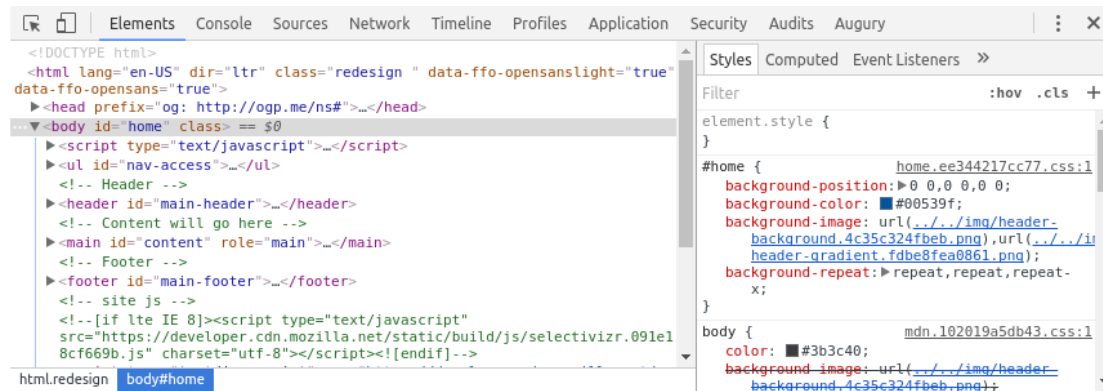
Augury

Augury es una extensión para el navegador que nos permite ver los componentes de Angular en el navegador, es decir, ofrece más información sobre los componentes que el inspector de elementos, como por ejemplo. la relación entre ellos, las rutas, etc.

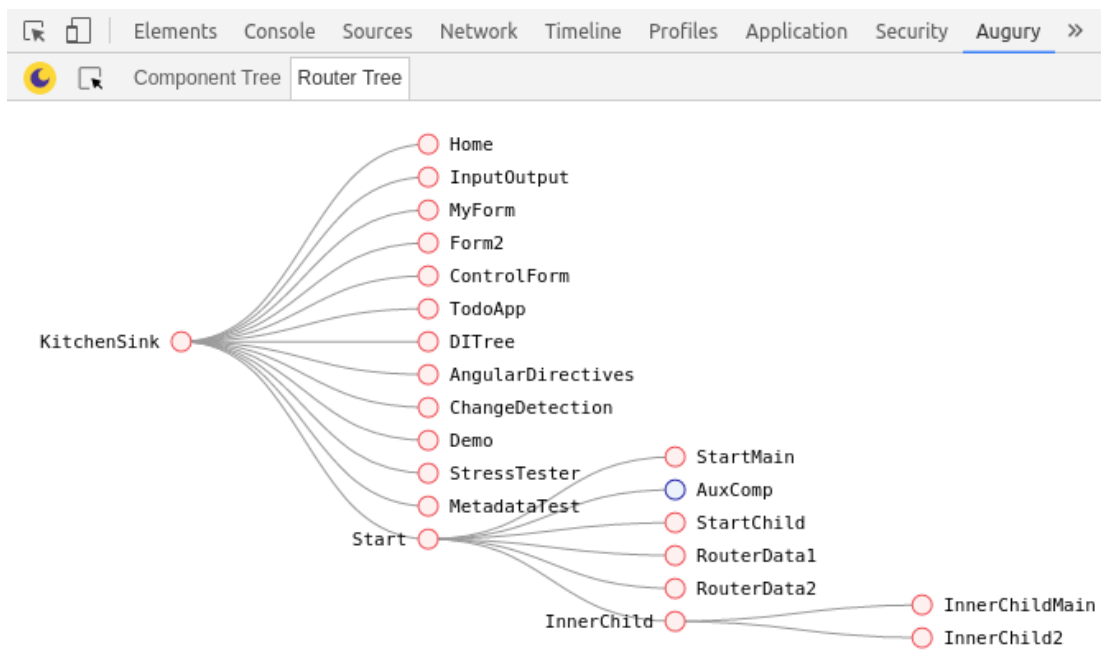
Puedes descargar augury desde su página oficial: [Página oficial de Augury](#)



Una vez instales Augury, aparecerá una nueva pestaña en el inspector de elementos:



Una funcionalidad de esta extensión que personalmente utilizo mucho es la posibilidad de ver un gráfico con todas las rutas de la página:



En páginas grandes esto permite tener un control del flujo de la aplicación para poder depurar y detectar errores. Además esta información puede resultar útil para decidir que rutas hacer que carguen de forma perezosa para ahorrar recursos.

debugger

Existe un comando especial para Javascript muy útil que también se puede aplicar en Angular. Se trata de la instrucción **debugger**, que permite para el código en el punto que queramos. Esto quiere decir que poniendo este comando vas a poder parar la ejecución justo antes de un error o de una parte en particular que quieras ver con más detenimiento. Su sintaxis es muy fácil de recordar:

```
debugger;
```

Y listo, si ahora ejecutas la aplicación y al abres en el navegador, su ejecución se parará justo al llegar a esta instrucción y por tanto con la consola podrás ver el valor de cada variable en ese instante.

Usos de ng-template, ng-container y ngTemplateOutlet

ng-template

Esta etiqueta sin saberlo ya la has usado antes, en concreto cuando creas un `ngIf` o un `ngFor`.

La etiqueta, como su nombre indica, sirve para representar una plantilla a renderizar en el componente. Esta plantilla o template, se puede componer a su vez de otros templates.

Vemos un ejemplo:

```
@Component({
  selector: 'app-root',
  template: `
    <ng-template>
      <button class="tab-button"
        (click)="login()">{{loginText}}</button>
      <button class="tab-button"
        (click)="signUp()">{{signUpText}}</button>
    </ng-template>
  `})
```

En el componente de arriba creamos un template con un par de botones, pero si abres la página haciendo **serve** te darás cuenta de que no se renderiza nada. Eso es porque esta etiqueta solo funciona con **directivas estructurales** (**ngIf**, **ngFor**, **NgSwitch**...)

Vemos como Angular utiliza esta etiqueta como azúcar sintáctico:

```
<div class="lessons-list" *ngIf="lessons else loading">
  ...
</div>
```

Al compilarse creará:

```
<ng-template [ngIf]="lessons">
  <div class="lessons-list">
    ...
  </div>
</ng-template>
```

Es mucho más sencillo poner simplemente el **ngIf** que andar creando un elemento por

fuera cada vez. Esto quiere decir que los **ng-template** quieras o no se usan en toda la aplicación todo el rato.

Un uso típico que tiene **ng-template** es el de poner crear **elses**, es decir, si queremos hacer:

```
<div *ngIf="loading">Dentro del IF</div>
<div *ngIf="!loading">Dentro del ELSE</div>
```

Otra manera de hacerlo con templates sería tal que así:

```
<div *ngIf="loading; else showItem2">Dentro del IF</div>

<ng-template #showItem2>
Dentro del ELSE
</ng-template>
```

Mediante template syntax creamos una variable y se la pasamos al elemento de abajo para que la muestre o no.

ng-container

Esta etiqueta la encuentro personalmente muy útil. Sirve para no tener en el DOM elementos de más. Es decir, por ejemplo si quieres poner un IF dentro de la vista, lo que normalmente harías es meterlo dentro de un `<div>` o similar:

```
<div *ngIf="users">
  <div class="users" *ngFor="let user of users">
    {{user}}
  </div>
</div>
```

Con esto se crea un elemento `div` exterior (el que crea el `ngIf`) que realmente no es necesario, si sustituimos el **div** exterior por un **ng-container** no tendremos este problema:

```
<ng-container *ngIf="users">
  <div class="users" *ngFor="let user of users">
    {{user}}
  </div>
</ng-container>
```

Con el código anterior solo se generará el **div de dentro**.

ngTemplateOutlet

Con la directiva ngTemplateOutlet puedes usar plantillas creadas con ng-template para usarlas en cualquier parte de la vista:

```
<ng-template #greet><span>Hello</span></ng-template>
```

```
<ng-container *ngTemplateOutlet="greet"></ng-container>
```

Podemos ver aquí cómo ng-container ayuda en este caso de uso: lo estamos usando para instanciar en la página la plantilla de carga que definimos anteriormente.

Nos referimos a la plantilla de carga a través de su referencia de plantilla #greet, y estamos utilizando la directiva estructural ngTemplateOutlet para instanciar la plantilla.

Podríamos añadir tantas etiquetas ngTemplateOutlet a la página como quisiéramos, e instanciar un número de plantillas diferentes. El valor pasado a esta directiva puede ser cualquier expresión que evalúe en una plantilla de referencia.

Cómo usar SASS con Angular

Para el desarrollo web SASS se ha convertido en una herramienta fundamental. Aunque actualmente CSS se sigue expandiendo (ofreciendo variables por ejemplo), todavía quedan cosas para llegar al nivel de SASS.

SASS también se puede usar con Angular, y si todavía no has creado el proyecto con Angular CLI, lo tienes muy fácil, tan solo tienes que ejecutar:

```
ng new my-sassy-app --style=scss
```

Puedes cambiar el `style=scss` por **sass** o **less** dependiendo de la variante que quieras.

Si ya tienes el proyecto creado de antes y usas CSS, cambiarlo a SASS puede ser un poquito más tedioso. Vamos a ver el procedimiento:

Lo primero es decirle a Angular que empiece a compilar SASS para ello:

```
ng set defaults.styleExt scss
```

Este comando añadirá al fichero **.angular-cli.json** lo siguiente:

```
"defaults": {  
  "styleExt": "scss",  
  "component": {  
  }  
}
```

Hay un problema con esto, los ficheros que ya hayas creado con sass no se van a renombrar. Si decides renombrarlos manualmente, te tienes que acordar en los componentes de actualizar el nombre también.

Por eso vengo con este script para que el proceso sea automático. Lo primero, buscar y reemplazar el nombre de todos los ficheros:

```
find . -name "*.css" -exec bash -c 'mv "$1" "${1%.css}".scss' - '{}' \;
```

Y por último actualizar todas las referencias a ficheros css para usar los scss:

```
find ./src -name "*.ts" -exec sed -i -e 's/\\(.*styleUrls.*\\)\\.css\\(.*\\)/\\1\\.scss\\2/g' {} +  
rm -r *.ts-e
```

Como en PDF no deja copiar y pegar, te dejo la dirección del script en github:

<https://gist.github.com/Frostqui/9fefbf424c887b5a07e773adecc52e8c>

Listo, ya puedes usar sass en tus proyectos. Esto no quiere decir que sea obligatorio usarlo, aunque yo personalmente lo recomiendo, sobre todo para tener que escribir menos código CSS.

Decoradores Angular

Antes de empezar directamente con los decoradores hay que aclarar que existen varios tipos de decoradores:

- Decoradores de clase, por ejemplo `@Component`
- Decoradores de propiedades y atributos, por ejemplo `@Input`
- Decoradores de métodos, por ejemplo `@HostListener`

Decoradores de clase

Con los decoradores indicas qué tipo de clase vas a declarar. Básicamente existen dos decoradores de clase, `@Component` y `@Module`

Por ejemplo:

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

```
@Component({  
  selector: 'example-component',  
  template: '<div>Example component!</div>',  
})
```

```
export class ExampleComponent {  
  constructor() {  
    console.log('Hey I am a component!');  
  }  
}
```

```
@NgModule({  
  imports: [],  
  declarations: [],  
})  
  
export class ExampleModule {  
  constructor() {  
    console.log('Hey I am a module!');  
  }  
}
```

```
}  
}
```

La diferencia entre componente y módulo es que los componentes controlan la vista de la aplicación y los módulos no.

Un módulo se puede componer de varios componentes. Como su nombre indica, sirve para hacer tu aplicación Angular modular declarando los componentes necesarios.

Decoradores de atributos y parámetros

Estos decoradores van dentro de la clase, en sus propiedades y parámetros. Estoy seguro de que estos decoradores ya los usado con Angular, por ejemplo:

```
import { NgModule, Component } from '@angular/core';  
  
@Component({  
  selector: 'example-component',  
  template: '<div>Example component!</div>',  
})  
export class ExampleComponent {  
  
  @Input() exampleProperty: string;  
  
  constructor() {  
    console.log('Hey I am a component!');  
  }  
}
```

Por debajo, Angular hace la magia de usar el decorador @Input para generar lo necesario para que puedas pasar atributos desde fuera del componente.

Los atributos de parámetro se usan dentro de los constructores. Estos decoradores también son muy típicos:

```
import { Component, Inject } from '@angular/core';  
import { MyService } from './my-service';  
  
@Component({  
  selector: 'example-component',  
  template: '<div>Example component!</div>'
```



```

})

export class ExampleComponent {
  constructor(@Inject(MyService) myService) {
    console.log(myService); // MyService
  }
}

```

En este caso se usa para hacer inyección de dependencias de los servicios en los componentes.

Decoradores de métodos

Como te podrás imaginar estos se usan junto a los métodos. Un ejemplo de estos decoradores es el `@HostListener` que sirve para capturar eventos, por ejemplo:

```

import { Component, HostListener } from '@angular/core';

@Component({
  selector: 'example-component',
  template: '<div>Example component!</div>'
})
export class ExampleComponent {
  @HostListener('click', ['$event'])
  onHostClick(event: Event) {
    // clicked, `event` available
  }
}

```

Muy bien, hemos visto los tipos de decoradores que hay, pero seguro que ya los conocías todos. Vamos a lo realmente interesante, cómo crear los tuyos propios.

Cómo crear tus propios decoradores

Lo que no mucha gente sabe de Angular, es que se pueden crear decoradores. Creando los tuyos propios vas a agilizar mucho el desarrollo de aplicaciones de Angular porque los vas a definir una sola vez y los vas a poder reutilizar en muchos sitios.

Los decoradores no son más que funciones que se ejecutan cuando Angular llama al decorador. La función puede recibir como parámetro la clase que se ha decorado (si es un decorador de clase), el atributo o parámetro o el método si es un decorador de método.

Vamos a crear un decorador que simplemente imprima por pantalla:

```
function Console(target) {  
    console.log('La clase decorada es: ', target);  
}
```

Para usarlo simplemente tienes que:

```
@Console  
class ExampleClass {  
    constructor() { }  
}
```

Yo recomiendo crear los decoradores en un fichero aparte para tenerlos localizados. En ese caso tienes que importar el decorador para poder usarlo en el componente.

Otra ventaja de los decoradores es que se le pueden pasar parámetros, en ese caso tienes que adaptar un poco la estructura del decorador:

```
function Console(message) {  
    // Message es el parámetro que le pasas al decorador  
    console.log(message);  
    return function(target) {  
        // Dentro de esta función recibes la clase o el método decorado  
        console.log('La clase decorada es: ', target);  
    };  
}
```

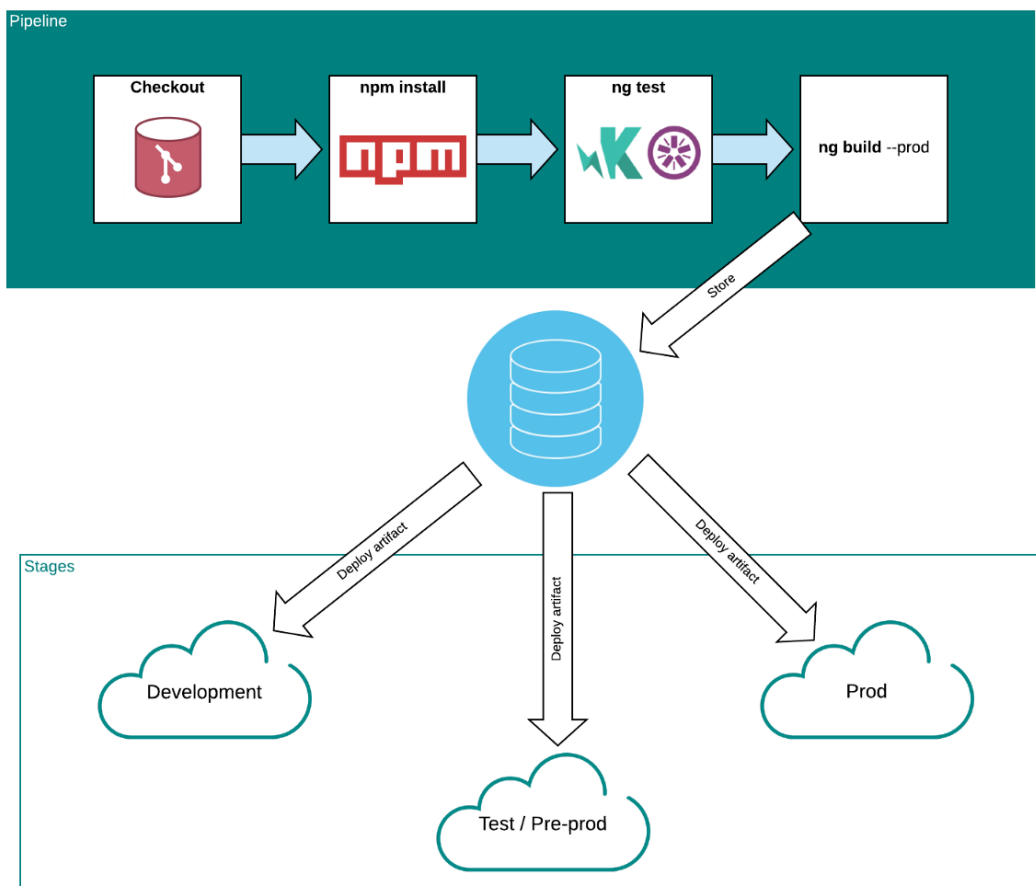
Y listo, ya puedes crear tus propios decoradores para reutilizar código entre componentes. Yo personalmente todavía no creo mis propios decoradores porque es algo que he aprendido recientemente, pero no descarto aplicarlos para nuevos proyectos.

Gestión de multiples entornos

Es muy común disponer de varios entornos de pruebas antes de publicar una aplicación Angular, sobre todo en entornos profesionales. Estos entornos se suelen conocer como **preproducción** y **producción**. El entorno de producción es el entorno que utilizan los usuarios, es decir, en una web, el entorno de producción es la maquina donde está desplegada la web que ve la gente. Los entornos de preproducción se utilizan para hacer pruebas antes de publicar algo que puedan ver los usuarios o clientes de nuestra aplicación.

Usualmente el entorno de preproducción se encuentra aislado al entorno de producción. Este entorno se trata de una copia del entorno de producción para poder testear toda la funcionalidad como si se tratara del entorno real.

Si utilizas Angular y lo conectas a una API REST, puede darse el caso en el que la API también tenga entorno de preproducción, por eso en este capítulo vamos a ver cómo podemos hacer para conectarnos a la API de preproducción o de producción dependiendo de dónde desplaguemos la aplicación de Angular sin tener que tocar nada.



En el gráfico de arriba puedes ver un ejemplo de flujo, pero el que yo recomiendo es el siguiente:

- El entorno de CD / CI (Continuous Delivery / Continuous Integration) detecta un cambio en la rama develop, es decir, detecta que se ha subido algo nuevo a la rama develop y ejecuta el deploy. Como estamos en la rama develop, pasa los tests y sube al entorno de preproducción.
- El entorno de CD / CI detecta un cambio en master y esta vez pasa los tests y despliega en el entorno de producción.

Con este procedimiento tenemos que tener claro que a master se sube cada menos tiempo que a develop, y solo cosas que estén funcionando y estén propadas en el entorno de preproducción.

Muy bien, vamos a ver cómo se puede integrar esto con Angular.

Si create tu proyecto con Angular CLI, seguramente tengas en el proyecto un par de archivos, `environment.ts` y `environment.prod.ts`. Si no los tienes los puedes crear. Estos archivos sirven para configurar las variables que se van a usar en cada uno de los entornos.

Por ejemplo, vamos a configurar dos variables distintas para los endpoints de la API. Empecemos configurando la API que se va a utilizar en producción. Abrimos el archivo **`environment.prod.ts`** y añadimos:

```
export const environment = {  
  production: true,  
  apiEndpoint: 'https://example.com/api/v0/'  
};
```

Para configurar el endpoint en los entornos de preproducción y desarrollo simplemente en el archivo **`environment.ts`** añades:

```
export const environment = {  
  production: true,  
  apiEndpoint: 'https://dev.example.com/api/v0/'  
};
```

Los archivos son iguales, excepto que cambia la dirección del endpoint de la API.

Ahora para acceder al valor del endpoint desde los servicios:

```
import { environment } from '../environments/environment';
```

```
console.log(environment.apiUrl);
```

Si te fijas simplemente importamos **environment**, si en lugar de eso importas **environment.prod** el valor del endpoint de la API siempre tendrá el valor de la del entorno de producción. Importando solo **environment** le decimos a Angular que utilice un entorno o otro dependiendo de cómo hagamos el build.

Para que Angular sustituya bien la variable por el valor de la del entorno de producción, tienes que hacer el build así:

```
ng build --prod
```

Es decir, tienes que configurar el entorno de CI / CD para que en producción le añada el flag **--prod** al build.

Creación de más entornos

Si estos dos entornos se te quedan cortos no te preocupes porque puedes crear todos los que quieras. Por ejemplo, pongamos que queremos crear un entorno llamado **test** para hacer pruebas rápidas antes de pasar a preproducción. Lo primero que tienes que hacer es crear un nuevo archivo, al mismo nivel que los otros dos entornos, llamado **environment.tests.ts**.

A partir de la versión 6 de Angular, dentro del archivo **angular.json** puedes configurar que ficheros de variables se usen en los distintos entornos. Si lo abres, podrás ver que ya hay uno creado para producción. Para crear otro añádelo debajo de la misma manera:

```
"tests": {
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.tests.ts"
    }
  ],
  // ...
}
```

Con el **replace** indicas el fichero base, y con el **with** indicas qué fichero se va a usar para ese entorno.

Para hacer el build con el nuevo entorno, se hace así:

```
ng build --configuration=tests
```

Y listo, puedes añadir los entornos que quieras con las variables que quieras para cada uno de ellos. Haciendo el build con la configuración de entorno que quieres usar, no tendrás que cambiar nunca las variables a mano.

Aislamiento de estilos para los componentes

Angular ofrece por defecto el aislamiento de estilos, esto quiere decir que aunque pongas la misma clase a un elemento del html en distintos componentes, sus estilos no se van a compartir. Esto es muy útil para el desarrollo de componentes porque no tienes que usar prefijos para las clases para no entrar en conflicto con estilos de otros componentes. Para realizar esto lo que hacen los frameworks es asignar clases con un hash para que cada componente tenga clases únicas.

Para que funcione el aislamiento tienes que crear los componentes así:

...

```
@Component({  
  selector: 'app-navbar',  
  templateUrl: './navbar.component.html',  
  styleUrls: ['./navbar.component.scss']  
})
```

...

Con esto vas a conseguir que los estilos queden mucho más simples al poder reutilizar las clases. Otra de sus ventajas es la facilidad de compartir componentes entre proyectos, ya que siempre vas a tener la seguridad de que los estilos no van a chocar con los del nuevo proyecto.

Angular añade también dos nuevos selectores a los estilos:

:host

A veces queremos dar estilos al propio elemento HTML del componente, y no lo que haya dentro de él.

Digamos, por ejemplo, que queremos dar forma al propio componente **app-root**, añadiéndole, por ejemplo, un borde adicional.

No podemos hacerlo usando estilos dentro de su archivo asociado `app.component.css`, ¿verdad?

Esto se debe a que todos los estilos dentro de ese archivo se incluirán en los elementos de la plantilla, y no en el **elemento app-root externo** en sí.

Si queremos cambiar el estilo del elemento host del propio componente, necesitamos el selector especial `:host`.

```
:host {  
  border: 2px solid red;  
  display: block;  
  padding: 20px;  
}
```

::ng-deep

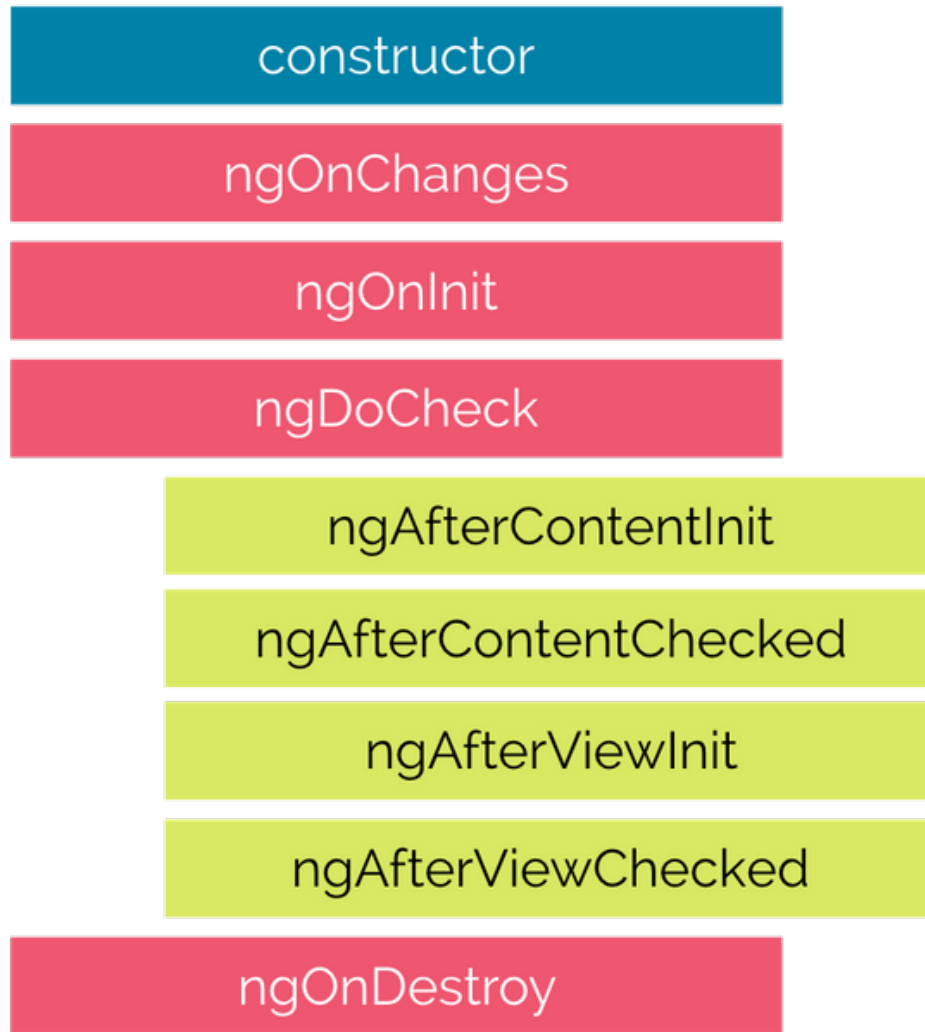
Si queremos que nuestros estilos de componentes pasen a todos los elementos hijos de un componente, pero no a ningún otro elemento de la página, podemos hacerlo combinando el `:host` con el selector `::ng-deep`

```
:host ::ng-deep h2 {  
  color: red;  
}
```

Esto aplicará estilo a todos los **h2** de **app-root** pero no a los de fuera suya. Es decir, es útil porque no tenemos que poner clases adicionales para dar estilo sino que directamente con **ng-deep** aplicamos a todos pero no a los de toda la aplicación web.

Ciclo de vida de los componentes

Sabes cómo se crean los componentes, pero ¿sabes por qué estado pasan hasta que se destruyen?



De esta lista vamos a ver los que considero más importantes.

ngOnInit

Este seguro que ya la conocías y si no te lo presento. Lo primero que hace Angular es inicializar el componente, es decir, cuando abres una página lo primero que ejecuta es este método. Veámoslo con un ejemplo:

```

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

@Component({selector: 'list-cmp', template: `...`})
export class ListComponent implements OnInit {
  errorMessage: string;
  products: Product[];

  constructor (private productService: ProductService) {}

  ngOnInit() { this.getProducts(); }

  getProducts() {
    this.productService.getProducts()
      .subscribe(
        products => this.products = products,
        error => this.errorMessage = <any>error);
  }
}

```

Como ves, lo primero que hay que hacer es importar Component y ngOnInit de Angular. Al crear el componente haces que implemente de OnInit y dentro del componente llamas a ngOnInit con lo que quieres que se ejecute al principio, en este caso la llamada a getProducts(). El OnInit se suele usar mucho para este tipo de cosas, llamadas a servicios para inicializar lo que se quiere pintar en pantalla.

ngOnChanges

Este método se ejecuta cuando Angular detecta un cambio en uno de los @Input y antes que OnInit, es decir, cuando desde fuera se modifica la variable que se le pasa al componente hijo. Veamos un ejemplo:

```

import {Component, OnChanges} from '@angular/core';

@Component({selector: 'list-cmp', template: `...`})
export class ListComponent implements OnChanges {
  errorMessage: string;
  @Input() message: string;

  constructor (private productService: ProductService) {}
}

```

```

    ngOnChanges(changes: SimpleChanges) { console.log(changes.previousValue); }

}

```

En este ejemplo ya no se llama al servicio al iniciar el componente, sino que la variable viene desde fuera. Cuando cambie desde fuera el valor de esta variable, se ejecutará el método `ngDoCheck`.

ATENCIÓN Se recomienda no usar `DoCheck` y `OnChanges` en el mismo componente

Este método tiene una particularidad, y es que recibe como parámetro un objeto de tipo `SimpleChanges`. Este objeto contiene la información del cambio, es decir, qué valor tenía antes y el nuevo valor que recibe.

```

class SimpleChange {
  constructor(previousValue: any, currentValue: any)
  previousValue : any
  currentValue : any
  isFirstChange() : boolean
}

```

AfterViewInit

Este método se ejecuta cuando Angular ya ha inicializado y cargado la vista. Este método te puede interesar si necesitas que pase algo o se ejecute algo tras haber cargado la vista. Por ejemplo para realizar una petición en segundo plano con la vista ya cargada.

```

import {Component, AfterViewInit} from '@angular/core';

@Component({selector: 'list-cmp', template: `...`})
export class ListComponent implements AfterViewInit {
  errorMessage: string;
  @Input() message: string;

  constructor (private productService: ProductService) {}

  ngAfterViewInit() { console.log("La vista ya se ha cargado"); }
}

```

OnDestroy

Como su nombre indica se ejecuta cuando el componente se destruye, es decir, cuando ya no está cargado, por ejemplo justo cuando abandonas una página, se descargan los componentes de la página en la que estaban y se cargan los de la página nueva.

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

@Component({selector: 'list-cmp', template: `...`})
export class ListComponent implements OnDestroy {
  errorMessage: string;

  @Input() message: string;

  constructor (private productService: ProductService) {}

  ngOnDestroy() { console.log("Descargando el componente"); }
}
```

Este método te puede servir para resetear estados, guardar cambios, etc.

Directivas

Las directivas en Angular son muy interesantes, permiten modificar el DOM y sus estilos. Hay 3 tipos de directivas:

- Componentes: Los componentes que has visto hasta ahora son directivas también porque cambian el comportamiento del DOM
- Estructurales: Añaden y eliminan elementos del DOM
- De atributo: Cambian los estilos y atributos de elementos del DOM o de otras directivas

Directivas de atributo

Las directivas de atributo se crean muy parecido a un componente normal. Simplemente necesitas poner `@Directive` para declarar el selector que va a tener en la vista y un constructor desde el que recibes el elemento del DOM sobre el que se aplica. Veamos un ejemplo:

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

Esta es la estructura básica de una directiva de atributo en Angular. Dentro del selector he puesto el nombre que va a tener la directiva para poder referenciarla, en este caso la he llamado poniendo `app` delante para que su nombre no entre en conflicto con selectores de librerías.

Esta directiva por el momento no hace nada, veamos un ejemplo de funcionalidad:

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
```

```

        el.nativeElement.style.backgroundColor = 'yellow';
    }
}

```

Dentro del constructor se declara el elemento perteneciente a ElementRef en el que vendrá el componente sobre el que se aplica la directiva. Dentro del constructor simplemente llamamos a su nativeElement para tener el elemento del DOM y modificamos su estilo.

Ahora si importas esta directiva que acabas de crear en un componente y se lo aplicas en el DOM de esta forma, por ejemplo:

```
<p appHighlight>Highlight me!</p>
```

Si ahora haces **ng serve** verás que el elemento sobre el que lo has aplicado tiene el fondo amarillo. Aparte de poder modificar los estilos del DOM, también podemos hacerlo dependiendo de si se produce un evento determinado, por ejemplo:

```

import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @Input('appHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'red');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}

```

```
}
```

En este ejemplo más complejo, hacemos uso del `@HostListener` para poder reaccionar a los eventos producidos por el ratón, en este caso cuando estamos sobre el elemento. Además, hay un atributo `@Input` para poder pasar el color desde el elemento sobre el que se va a usar.

Para usarlo:

```
<p appHighlight highlightColor="yellow">Highlighted in yellow</p>
```

Este ejemplo es muy sencillo y tiene poca utilidad pero te puedes hacer una idea de las posibilidades que ofrece esto. Puedes crear una serie de directivas con efectos reaccionando a eventos para que las puedas reutilizar a lo largo de todo el proyecto.

Puedes ver la demo de la directiva aquí

Directivas estructurales

Las directivas estructurales están a cargo del DOM, pueden modificar sus elementos, añadir, borrar, etc. Puedes reconocer una directiva estructural porque para usarlas pones un asterisco `*` y después su nombre. Sin darte cuenta llevas mucho tiempo usando directivas estructurales, pero no te has dado cuenta. Por ejemplo `*ngIf="condicion"` es una directiva estructural que se encarga de mostrar o no el elemento sobre el que se aplica la directiva dependiendo de la condición. La directiva de `ngIf` no muestra y oculta los elementos sobre los que se aplica sino que físicamente los borra y los crea dinámicamente.

Los nombres de las directivas estructurales se suelen poner con la convención `lower-CamelCase`, es decir, empiezan en minúsculas y cada palabra se pone junto a la anterior en mayúsculas la primera letra. El asterisco se pone porque es azúcar sintáctico, es decir, realmente por debajo Angular cuando llega al asterisco, lo cambia por un template, de esta forma:

```
<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById" [class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>
```

```
<!-- Angular lo traduce a: -->
```

```
<ng-template ngFor let-hero [ngForOf]="heroes" let-i="index" let-odd="odd" [ngForTrackBy]="trackBy
```

```

    <div [class.odd]="odd">({{i}}) {{hero.name}}</div>
</ng-template>

```

La estructura básica de una directiva estructural es muy similar a las anteriores:

```

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({ selector: '[appUnless]' })
export class UnlessDirective {
}

```

Como en las otras directivas, se define su selector entre corchetes. Su selector será la forma de llamar a la directiva desde la vista. Una particularidad de esta directiva es que dentro del constructor define dos objetos:

```

constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef)

```

En el templateRef llega la estructura del DOM sobre el que se aplica la directiva, y en el viewContainer se genera la nueva estructura del DOM. Además esta directiva necesita de un atributo @Input llamado appUnless con un setter para poner true o false. La estructura quedaría de la siguiente manera:

```

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

/**
 * Add the template content to the DOM unless the condition is true.
 */
@Directive({ selector: '[appUnless]' })
export class UnlessDirective {
    private hasView = false;

    constructor(
        private templateRef: TemplateRef<any>,
        private viewContainer: ViewContainerRef) { }

    @Input() set appUnless(condition: boolean) {
        if (!condition && !this.hasView) {
            this.viewContainer.createEmbeddedView(this.templateRef);
            this.hasView = true;
        }
    }
}

```



```

    } else if (condition && this.hasView) {
        this.viewContainer.clear();
        this.hasView = false;
    }
}
}

```

Como ves, si la condición es falsa se crea el viewContainer con la vista y si es true simplemente se limpia la vista de tal forma que no se renderice.

Si importas esta directiva y la usas en el HTML verás que se obtiene un funcionamiento muy parecido que con un *ngIf

```

<p *appUnless="condition" class="unless a">
    Este párrafo se muestra si la condición es falsa
</p>

```

Observables y promesas

Los observables permiten el paso de información y mensajes entre los que los publican y los que están suscritos a esos mensajes. Un ejemplo de uso de Observables es el paso de información entre los servicios y los componentes.

Para usar los observables tienes que importarlos así:

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

Los mensajes son lazy (perezosos), eso quiere decir que solo se envían a los que estén suscritos. Dentro del observable defines una función en la que se publican los valores pero que no se va a ejecutar hasta que no lo recibe un consumidor, es decir, hasta que el consumidor ejecuta `suscribe()`. Veamos un ejemplo con un productor:

```
// Esta función se ejecuta cuando el consumidor ejecuta susbscribe()
function sequenceSubscriber(observer) {
  // Con la función next() vamos sacando valores
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  return {unsubscribe() {}};
}

// Creación del Observable con la función que se ejecuta en el consumidor
const sequence = new Observable(sequenceSubscriber);

// Nos subscribimos al observable y vamos imprimiendo los valores que recibimos
sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});
```

Con este ejemplo, el consumidor se suscribe y con la función `next()` va imprimiendo los valores que le llegan del observable. Dentro de la función `sequenceSubscriber` publicamos los valores asíncronamente.

Veamos un ejemplo aplicado a peticiones http en los servicios de Angular:

```

getSeeschweiler(): Observable<any>{
  return this.http.get('https://api.github.com/users/seeschweiler');
}

```

Para este ejemplo lo que hago es hacer una llamada a la api de github. Como ves, devuelvo en la llamada un Observable de tipo any, aunque lo suyo es crear una interfaz con el modelo de datos que venga desde la api. En este caso, la función que se va a ejecutar en el subscribe() del consumidor es la llamada http devolviendo el valor de la petición.

Ahora, desde el componente:

```

this.service.getSeeschweiler().subscribe(
  data => {
    this.data = data;
    console.log(data);
  }
);

```

Al hacer subscribe() se ejecuta la llamada http del servicio y se devuelve el valor de la petición.

Una buena práctica para el uso de los Observables es usar el objeto Subscription para tener un control de la subscripción al observable. Esto es recomendable para poder poner en el OnDestroy una llamada para desuscribirte y así evitar pérdidas de memoria:

```

import { Component, OnInit, OnDestroy } from '@angular/core';

```

```

import { ModelService } from '../model.service';

```

```

import { Subscription } from 'rxjs/Subscription';

```

```

@Component({
  selector: 'app-model',
  templateUrl: '../model.component.html',
  styleUrls: ['../model.component.css']
})

```

```

export class ModelComponent implements OnInit, OnDestroy {

```

```

  private _modelSubscription: Subscription;

```

```

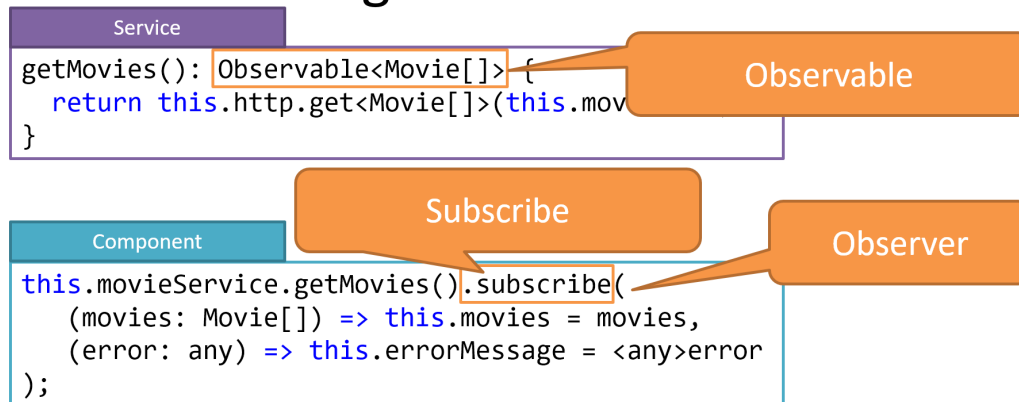
constructor(private modelService: ModelService) { }

ngOnInit() {
  this.modelSubscription = this.modelService.getUsers().subscribe(users => {
    console.log(users);
  })
}

ngOnDestroy() {
  if (this.modelSubscription) {
    this.modelSubscription.unsubscribe();
  }
}
}

```

Using an Observable



Animaciones

Aunque puedes crear tus propias animaciones con CSS, o incluso tus propios componentes con Animaciones, Angular tiene una forma nativa de crear animaciones. Además de poder crearlas, ofrece maneras de poder lanzarlas y controladas con determinados eventos, por ejemplo, cuando se muestran los elementos de una listas, que se vayan lanzando las animaciones escalonadamente.

Para usar las animaciones tienes que importar su módulo en el app.module.ts:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule
  ],
  declarations: [ ],
  bootstrap: [ ]
})
export class AppModule { }
```

Simplemente tienes que importar en el componente lo que vayas a usar de estas funciones:

```
import { Component, HostBinding } from '@angular/core';
import {
  trigger,
  state,
  style,
  animate,
  transition,
  // ...
} from '@angular/animations';

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

```

styleUrls: ['app.component.css'],
animations: [
    // Aquí metes las animaciones
]
})

```

Ahora que tienes activadas las animaciones en Angular, las puedes usar importando en cada componente lo que vayas a usar. Angular ofrece lo siguiente:

- **trigger():** Inicia la animación y sirve como contenedor para todas las demás.
- **style():** Define uno o más estilos de CSS para usar en animaciones.
- **state():** Crea un conjunto de estilos CSS con nombre que deben aplicarse en una transición a un estado dado.
- **animate():** Especifica la información de tiempo para una transición. Puede contener llamadas `style()` dentro.
- **transition():** Define la secuencia de animación entre dos estados con nombre.
- **keyframes():** Permite un cambio secuencial entre estilos dentro de un intervalo de tiempo especificado.
- **group():** Especifica un grupo de pasos de animación (animaciones internas) que se ejecutarán en paralelo.
- **query():** Se usa para encontrar uno o más elementos HTML internos dentro del elemento actual.
- **sequence():** Especifica una lista de pasos de animación que se ejecutan secuencialmente, uno por uno.
- **stagger():** Escalona el tiempo de inicio de las animaciones para múltiples elementos.
- **animation():** Produce una animación reutilizable que puede ser invocada desde cualquier otro lugar. Usado junto con `useAnimation()`.
- **useAnimation():** Activa una animación reutilizable. Se utiliza con `animation()`.
- **animateChild():** Permite que las animaciones de los componentes hijo se ejecuten en el mismo tiempo que el padre.

Si te estás preguntando qué son los estados, Angular define los siguientes estados:

- **Void state:** Define el estado en el que se encuentra un elemento que no está en el DOM, es decir, por ejemplo, un elemento que se acaba de crear pero que no se ha pintado todavía en pantalla. Este estado se suele utilizar cuando quieres animar efectos de elementos apareciendo y desapareciendo de pantalla
- **Wildcard state:** El estado por defecto que tienen los elementos
- **Custom state:** Este estado lo puedes crear tu mismo según tus necesidades, tiene

que ser definido explícitamente en el código.

Veamos un ejemplo de animación en Angular. Por ejemplo vamos a crear una animación que cambie el tamaño de un elemento del DOM:

```
animations: [  
  trigger('changeDivSize', [  
    state('initial', style({  
      backgroundColor: 'green',  
      width: '100px',  
      height: '100px'  
    })),  
    state('final', style({  
      backgroundColor: 'red',  
      width: '200px',  
      height: '200px'  
    })),  
    transition('initial=>final', animate('1500ms')),  
    transition('final=>initial', animate('1000ms'))  
  ]),  
)  
]
```

Definimos nuestros propios estados con los estilos que queremos para ese estado. Debajo usamos el método `transition()` para que cree la animación entre ambos estados. Puedes ver que ya estamos utilizando 4 funciones que ofrece Angular para las animaciones: `trigger`, `state`, `transition` y `animate` para poder elegir la duración de la animación.

Para poder ver bien la animación vamos a crear un método para poder cambiar de estado:

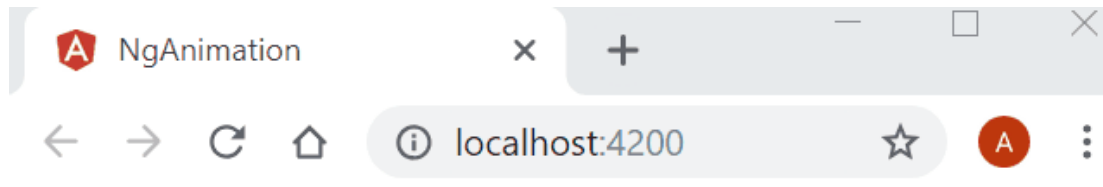
```
currentState = 'initial';  
  
changeState() {  
  this.currentState = this.currentState === 'initial' ? 'final' : 'initial';  
}
```

Por último creamos la vista con el botón para cambiar el estilo:

```
<h3>Change the div size</h3>  
<button (click)="changeState()">Change Size</button>  
<br />
```

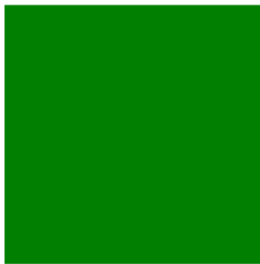
```
<div [@changeDivSize]=currentState></div>  
<br />
```

Si te fijas, en el botón llamamos al método para cambiar de estado al ser pulsado. Debajo, creamos el div en el que aplicar al animación con `[@changeDivSize]` y le pasamos la variable donde guardamos el estilo.



Change the div size

Change Size



Supongamos ahora que queremos crear una animación al añadir y quitar elementos de un array. Para ello existe una keyword especial que se usa al definir la animación para decidir los estilos al aparecer elementos y desaparecer:

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'],  
  animations: [  
    trigger('itemAnim', [  

```



```

    transition(':enter', [
      style({transform: 'translateX(-100%)'}),
      animate(350)
    ]),
    transition(':leave', [
      group([
        animate('0.2s ease', style({
          transform: 'translate(150px,25px)'
        })),
        animate('0.5s 0.2s ease', style({
          opacity: 0
        }))
      ])
    ])
  ])
})

```

En este caso no hace falta definir los estados como hemos hecho antes porque Angular ya los define (enter y leave). Primero creamos la animación al añadir elementos al array, para este ejemplo entrarán desde la izquierda. Para la animación de salida, utilizo group() para poder encadenar dos animaciones, una de desplazamiento y otra de opacidad.

En la vista ya podemos usar la animación que acabamos de crear:

```

<input #itemInput
  (keyup.enter)="addItem(itemInput.value); itemInput.value=''>

<button
  (click)="addItem(itemInput.value); itemInput.value=''>
  Add
</button>

<ul *ngIf="items">
  <li *ngFor="let item of items"
    (click)="removeItem(item)"
    [@itemAnim]>{{ item }}</li>
</ul>

```

El botón simplemente añade los elementos a la lista que creemos mediante el input y al hacer clic sobre ellos los borramos para que se muestre la animación de salida.

PWA



En los últimos tiempos se habla mucho de las webs PWA, pero ¿qué son?. PWA es el acrónimo de Progressive Web Application, es decir, son páginas webs progresivas que aparte de adaptarse a todo tipo de pantallas, ofrecen la ventaja de poder instalarse en dispositivos móviles. Actualmente Google permite subir webs PWA a la Play Store. Además otra de las ventajas es que permiten ser utilizadas sin conexión, ya que los recursos de la web quedan almacenados en la caché del navegador.

Para crear una web PWA necesitas varias cosas:

- Un archivo manifest.json en el que se declara la información de la web, iconos, etc.
- Un service worker, es decir, un código que se ejecuta en paralelo en el navegador encargado de guardar la página para servirla cuando no haya conexión

Crear una PWA con Angular es muy sencillo. Existe un paquete que precisamente crea un service worker por lo que solo necesitas crear un archivo manifest manualmente en la carpeta src. Veamos un ejemplo de manifest.json:

```
{  
  "short_name": "Nombre corto de tu web",  
  "name": "Nombre largo de tu web",  
  "icons": [  
      
  ]  
}
```

```

    {
      "src": "/assets/icon512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "start_url": "index.html",
  "background_color": "#FBFBFB",
  "theme_color": "#022E63",
  "display": "standalone",
  "orientation": "portrait"
}

```

Este ejemplo contiene los parámetros mínimos que se requieren en todo archivo manifest.json.

El icono y el nombre corto serán los que se muestren en la pantalla del dispositivo móvil al ser instalada la web.

Ya puedes instalar el paquete que crea el service worker, para ello:

```
npm install @angular/service-worker --save
```

Para que funcionen los service workers en Angular tienes que activar un flag en el fichero de configuración angular-cli.json:

```

"apps": [
  {
    "root": "src",
    "outDir": "dist",
    "assets": [
      "assets",
      "favicon.ico",
      "manifest.json",
      "service-worker.js"
    ],
    "index": "index.html",
    "main": "main.ts",
    "polyfills": "polyfills.ts",
    "test": "test.ts",

```

```

    "tsconfig": "tsconfig.json",
    "testTsconfig": "tsconfig.json",
    "prefix": "app",
    "serviceWorker": true,
    "styles": [
      "styles.css"
    ],
    ...

```

O directamente ejecutando sobre el proyecto:

```
ng set apps.0.serviceWorker=true
```

También tienes que importarlo en el app.module.ts:

```

import { ServiceWorkerModule } from '@angular/service-worker'
import { environment } from '../environments/environment';

...

@NgModule({
  imports: [
    ...
    ServiceWorkerModule.register('/ngsw-worker.js', {enabled: environment.production})
  ],
  ...
})
export class AppModule { }

```

Y por último crear el archivo de configuración del service worker, llamado ngsw-config.json:

```

{
  "index": "/index.html",
  "assetGroups": [{
    "name": "app",
    "installMode": "prefetch",
    "resources": {
      "files": [
        "/favicon.ico",
        "/index.html"

```

```

    ],
    "versionedFiles": [
      "/*.bundle.css",
      "/*.bundle.js",
      "/*.chunk.js"
    ]
  }
}, {
  "name": "assets",
  "installMode": "lazy",
  "updateMode": "prefetch",
  "resources": {
    "files": [
      "/assets/**"
    ]
  }
}]
}

```

Listo, ya tienes la pwa creada, automáticamente se generará con el comando:

```
ng build --prod
```

Si entras en la carpeta /dist/ te encontrarás con el fichero del service worker y el manifest si todo lo has creado bien.

Si has desplegado Angular tras hacer el build. al abrir la página web dos veces, dejando pasar 5 minutos, aparecerá un cartel preguntando si instalar la app, al aceptar, automáticamente se añadirá la web junto a las aplicaciones del móvil.

SEO en Angular



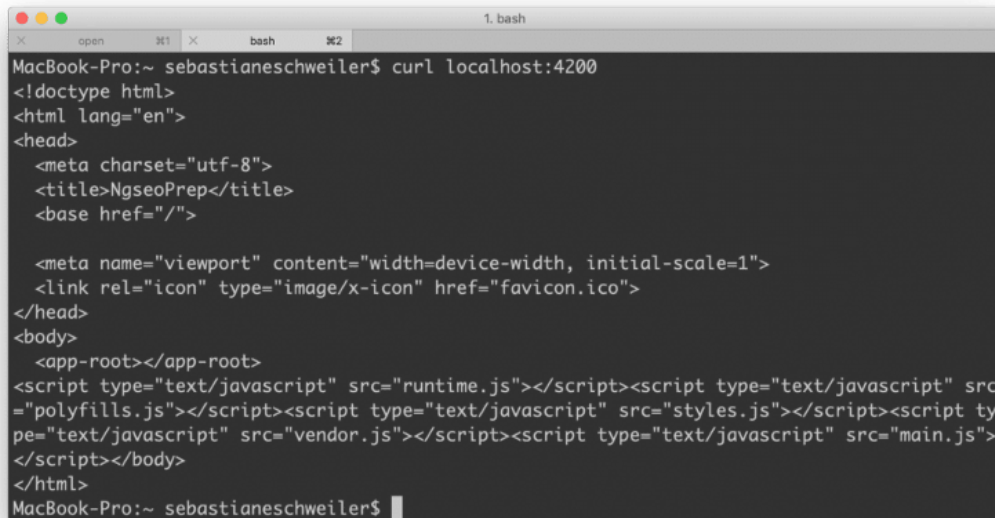
El SEO es una parte importantísima para cualquier negocio. El SEO se basa en conseguir que más gente visite tu página web o la de tu empresa. Hay dos tipos de SEO principalmente, SEO on page y SEO off page. El SEO on page se trata de todo lo que puedes hacer respecto al código de tu web para que genere más tráfico y es lo que vamos a ver en esta sección. El SEO off page, como su nombre indica es todo lo relacionado con tu negocio fuera de tu página. es decir, en redes sociales, publicidad, etc.

Como bien sabes, Angular es un framework para construir páginas SPA, es decir, todo el contenido se pinta de una vez al cargar la página. Esto tiene una desventaja, como la página se renderiza en el cliente que abre la web, para los motores de búsqueda no hay contenido. Haz la prueba, ejecuta:

```
ng serve
```

Y después, en otra ventana:

```
curl localhost:4200
```



```
MacBook-Pro:~ sebastianschweiler$ curl localhost:4200
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>NgseoPrep</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
<script type="text/javascript" src="runtime.js"></script><script type="text/javascript" src
="polyfills.js"></script><script type="text/javascript" src="styles.js"></script><script ty
pe="text/javascript" src="vendor.js"></script><script type="text/javascript" src="main.js">
</script></body>
</html>
MacBook-Pro:~ sebastianschweiler$
```

Esto es lo que los motores de búsqueda ven de tu página. No hay contenido sino que se genera mediante javascript al abrir la página y eso es un problema. ¿Cuál es la solución entonces? Server side rendering, es decir, necesitamos que la web se renderice en un servidor para servirla ya renderizada.

Afortunadamente para nosotros hay disponible un paquete de Angular para facilitar la transformación de una aplicación hecha con Angular a server side rendering.

Primero instala este paquete:

```
npm install -g @ng-toolkit/init
```

Si tienes la última versión de Angular cli ejecuta:

```
ng add @ng-toolkit/universal
```

Este comando añadirá todo lo necesario para transformar una aplicación angular a server side rendering.

A continuación ejecuta:

```
npm install
```

```
npm run build:prod
```

```
npm run server
```

Esta vez el servidor en local se inicia en <https://localhost:8080>

El resultado habrá cambiado respecto a lo que viste anteriormente:

```
MacBook-Pro:~ sebastianschweiler$ curl localhost:8080
<!DOCTYPE html><html lang="en"><head>
  <meta charset="utf-8">
  <title>NgseoPrep</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
<link rel="stylesheet" href="styles.3bb2a9d4949b7dc120a9.css"><style ng-transition="app-root-t"></style></head>
<body>
  <app-root _ngghost-sc0="" ng-version="6.1.9"><div _ngcontent-sc0="" style="text-align:center"><h1 _ngcontent-sc0=""> Welcome to ngeo-prep! </h1></div><h2 _ngcontent-sc0="">Here are some links to help you start:</h2><ul _ngcontent-sc0=""><li _ngcontent-sc0=""><h2 _ngcontent-sc0=""><a _ngcontent-sc0="" href="https://angular.io/tutorial" rel="noopener" target="_blank">Tour of Heroes</a></h2></li><li _ngcontent-sc0=""><h2 _ngcontent-sc0=""><a _ngcontent-sc0="" href="https://github.com/angular/angular-cli/wiki" rel="noopener" target="_blank">CLI Documentation</a></h2></li><li _ngcontent-sc0=""><h2 _ngcontent-sc0=""><a _ngcontent-sc0="" href="https://blog.angular.io/" rel="noopener" ta rget="_blank">Angular blog</a></h2></li></ul><router-outlet _ngcontent-sc0=""></router-outl et></app-root>
<script type="text/javascript" src="runtime.ec2944dd8b20ec099bf3.js"></script><script type="text/javascript" src="polyfills.f6ae3e8b63939c618130.js"></script><script type="text/javas cript" src="main.8a7e9e6eed30eaba457.js"></script>

<script id="app-root-state" type="application/json">{}</script></body></html>MacBook-Pro:~
```

Otro factor muy importante para el SEO es todo lo que puedes añadir en la etiqueta de tu página web. Como se trata de una web SPA, todas las páginas usan el mismo archivo html y es Angular el que se encarga de gestionar la vista, por lo que tendremos que actualizar las etiquetas del

Veamos un ejemplo:

48

```

@Component({
  selector: 'app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class HomeComponent {
  constructor(private meta: Meta) {
    this.meta.addTag({ name: 'twitter:card', content: 'summary_large_image' });
    this.meta.addTag({ name: 'twitter:site', content: '@codingpotions' });
    this.meta.addTag({ name: 'twitter:title', content: 'Blog de desarrollo web' });
    this.meta.addTag({ name: 'twitter:image', content: 'https://codingpotions.com' });
  }
}

```

En este ejemplo Angular añadirá las etiquetas para la vista previa en twitter en este componente, y como se trata del componente app y todos heredan de él entonces se incluirán también en cada página hija.

Angular también permite modificar una etiqueta meta que ya has creado:

```

this.meta.updateTag(
  { name: 'twitter:card', content: 'summary' },
  `name='twitter:card'`
);

```

Incluso permite borrar etiquetas:

```

constructor(private meta: Meta) {
  this.meta.removeTag('charset');
}

```

Ahora ya sabes cómo hacer que el contenido se renderice para los motores de búsqueda y cómo crear y actualizar las etiquetas del

, con eso te debería bastar para poder hacer SEO con Angular.

Cómo estructurar un proyecto grande con Angular

Uno de los problemas principales a la hora de mantener un proyecto grande en Angular es el cómo estructurarlo. Yo mismo al empezar con Angular no tenía claro cómo hacerlo y acababa con una estructura muy llosa.

Lo que vas a leer a continuación es lo que yo considero buenas prácticas y que a mi personalmente me ha funcionado, pero puede que para tu caso de uso no sea útil. Es por eso que te tienes que tomar esto como una guía que puedes adaptar a tus necesidades.

El concepto clave es que tienes que tener una estructura que te permita buscar rápidamente el código que necesitas cambiar. Tienes que empezar el proyecto teniendo al menos una idea de cómo quieres que termine el proyecto para poder tener una estructura acorde.

La estructura de carpetas que yo propongo es la siguiente (las carpetas se crean en `/src/app/`):

La carpeta core

```
|-- core
    |-- [+] authentication
    |-- [+] footer
    |-- [+] guards
    |-- [+] http
    |-- [+] interceptors
    |-- [+] mocks
    |-- [+] services
    |-- [+] header
    |-- core.module.ts
    |-- ensureModuleLoadedOnceGuard.ts
    |-- logger.service.ts
```

La carpeta core contiene funcionalidades y servicios únicos en la aplicación, es decir, por cada funcionalidad o servicio hay una sola instancia en toda la aplicación. Como ves arriba, un ejemplo típico es poner en esta carpeta todo lo que tiene que ver con la autenticación de usuarios, o por ejemplo el footer si solo hay uno en toda la web.

Aquí también puedes añadir servicios, en este caso yo suelo meter los servicios globales para la aplicación ya que lo suyo es que cada módulo de la aplicación gestione sus servicios.

Otro detalle a tener en cuenta es que he añadido el archivo **code.module.ts**. Para facilitar la modularidad de la aplicación se recomienda separar también los módulos, es decir, los archivos en los que se importan los componentes y servicios que se vayan a usar.

En esta carpeta también se suelen colocar los **guards** que como sabes, sirven para proteger rutas y páginas a ciertos usuarios que puedes definir.

La carpeta shared

```
|-- shared
    |-- [+] components
    |-- [+] directives
    |-- [+] pipes
    |-- shared.module.ts
```

Como su nombre indica, en esta carpeta se encuentran todo lo que se pueda compartir a lo largo de toda la aplicación. Por ejemplo, dentro de la carpeta components puedes colocar botones, spinners, iconos, etc.

Además, esta carpeta sirve para alojar directivas y pipes ya que también se pueden usar en otros componentes de la web.

La carpeta pages

```
|-- pages
    |-- [+] page1
    |-- [+] page2
    |-- [+] page3
    |-- pages.module.ts
```

Aquí suelo colocar las vistas de la aplicación, es decir, los componentes encargados de cada página de la web. Cada componente aquí colocado tiene asociado una vista en la aplicación, de esta forma tengo agrupadas todas las páginas que van a componer el proyecto.

En el ejemplo he llamado a las carpetas page1, page2 y page3 pero lo suyo es que las nombres con el nombre que tiene la página para que sean fáciles de localizar.

La carpeta components

```
|-- components
```

```
|-- [+] component1  
|-- [+] component2  
|-- [+] component3  
|-- components.module.ts
```

Este es el cajón desastre por así decirlo, aquí meto todo lo que no va dentro de las otras carpetas. Como en el apartado anterior, lo suyo es que cada carpeta tenga el nombre del componente que va dentro para que sea todo más fácil de localizar

Otras consideraciones

Mucha gente también suele crear una carpeta llamada UI en la que coloca componentes únicamente de diseño, es decir, componentes que no siguen una lógica de negocio y que son puramente estéticos. No la suelo usar porque al final en los proyectos que yo hago no termino con tantos componentes de diseño. No me parece mala idea y si a ti te parece que tiene sentido en tu proyecto puedes usarlo sin problemas.

Eso es todo en cuanto a estructura de proyectos grandes, como he dicho utiliza lo que a ti te venga mejor en el proyecto que estás desarrollando, siempre tienes que adaptar las guías a las necesidades de tu propio proyecto. Intenta que todo sea modular, separado entre sí y con nombres que te sean fáciles de buscar y con eso no vas a tener problemas en proyectos grandes.

Lazy components

¿Sabes lo que significa laziness en el frontend? Cargar algo de forma lazy significa que solo se carga cuando se necesita. Con Angular crear aplicaciones que se cargan al inicio para que al navegar por las páginas el usuario no tenga que esperar? pero ¿no sería fantástico que pudieras decidir qué quieres cargar al inicio y que no? Por ejemplo te interesa que ciertas páginas que no se suelen visitar no estén siempre cargadas y así hacer que la primera carga de la web sea más rápida.

Afortunadamente crear componentes lazy en Angular es bastante sencillo, veamos un ejemplo:

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

const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'lazymodule', loadChildren: './lazymodule/lazymodule.module#LazyModuleModule' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})

export class AppRoutingModule { }
```

En este caso la ruta **lazymodule** se cargará cuando navegue hasta ella pero no antes. Fíjate en que en este caso hay que crear un atributo **loadChildren**. También a la hora de poner la ruta al fichero es distinto. Tienes que indicar la ruta a un fichero module, que será el encargado de declarar el componente. Además, tienes que indicar seguido de **#** el nombre dentro del archivo module.

Si echamos un vistazo al fichero module será algo tal que así:

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

import { Component1Component } from './component1/component1.component';
import { Component2Component } from './component2/component2.component';
import { Component3Component } from './component3/component3.component';

const routes: Routes = [
```

```

    { path: '', component: Component1Component },
    { path: 'component2', component: Component2Component },
    { path: 'component3', component: Component3Component },
  ];

@NgModule({
  imports: [
    RouterModule.forChild(routes)
  ],
  declarations: [Component1Component, Component2Component, Component3Component]
})
export class LazyModuleModule {}

```

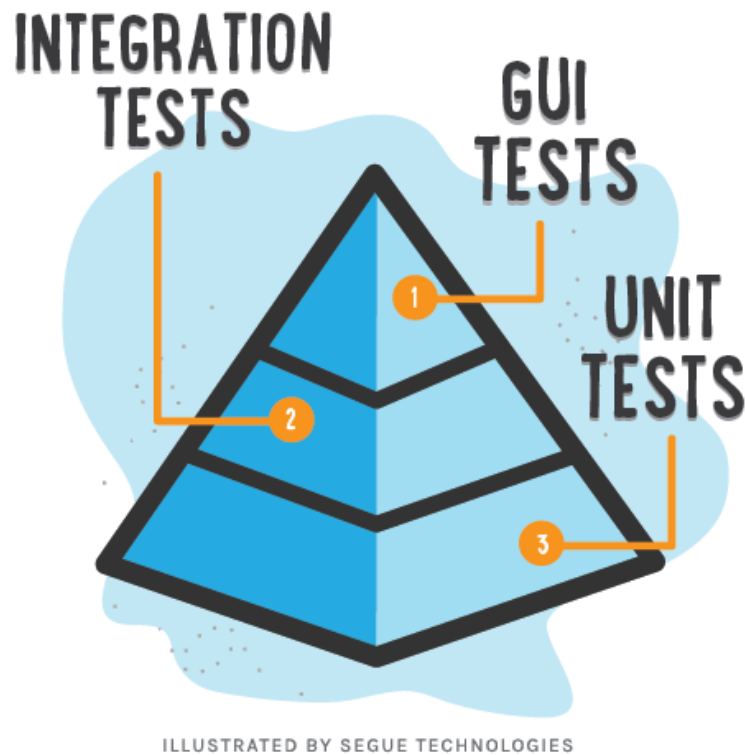
Testing unitario

Los tests son una pieza fundamental en los proyectos de hoy en día. Si tienes un proyecto grande es esencial tener una buena suite de tests para poder probar la aplicación sin tener que hacerlo manualmente. Además si lo combinas con la integración continua puedes minimizar el riesgo y los bugs futuros.

Antes de meternos de lleno en testear aplicaciones Angular, es importante saber qué tipos de tests que existen:

- **Tests Unitarios:** Consiste en probar unidades pequeñas (componentes por ejemplo).
- **Tests End to End (E2E):** Consiste en probar toda la aplicación simulando la acción de un usuario, es decir, por ejemplo para desarrollo web, mediante herramientas automáticas, abrimos el navegador y navegamos y usamos la página como lo haría un usuario normal.
- **Tests de Integración:** Consiste en probar el conjunto de la aplicación asegurando la correcta comunicación entre los distintos elementos de la aplicación. Por ejemplo, en Angular observando cómo se comunican los servicios con la API y con los componentes.

En este artículo vamos a cubrir el **testeo unitario en Angular**.



Tests unitarios con Jasmine

Para hacer tests unitarios en Angular se suele usar Jasmine. Jasmine es un framework Javascript (No es exclusivo de Angular, lo puedes usar en cualquier aplicación web), para la definición de tests usando un lenguaje natural entendible por todo tipo de personas.

Un test en Jasmine tiene esta pinta:

```
describe("A suite name", function() {  
  it("contains spec with an expectation", function() {  
    expect(true).toBe(true);  
  });  
});
```

- **describe:** Define una suite de tests, es decir, una colección de tests. Ésta función recibe dos parámetros, un string con el nombre de la suite y una función donde definiremos los tests.
- **it:** Define un test en particular. Recibe como parámetro el nombre del test y una función a ejecutar por el test.

- **expect**: Lo que espera recibir el test. Es decir, con expect hacemos la comprobación del test. Si la comprobación no es cierta el test falla. En el ejemplo anterior comprobamos si true es true luego el test pasa. Cómo ves no podemos simplemente hacer la comprobación haciendo poniendo la operación ===, tenemos que usar las funciones que vienen con Jasmine, las cuales son:

- expect(array).toContain(member);
- expect(fn).toThrow(string);
- expect(fn).toThrowError(string);
- expect(instance).toBe(instance);
- expect(mixed).toBeDefined();
- expect(mixed).toBeFalsy();
- expect(mixed).toBeNull();
- expect(mixed).toBeTruthy();
- expect(mixed).toBeUndefined();
- expect(mixed).toEqual(mixed);
- expect(mixed).toMatch(pattern);
- expect(number).toBeCloseTo(number, decimalPlaces);
- expect(number).toBeGreaterThan(number);
- expect(number).toBeLessThan(number);
- expect(number).toBeNaN();
- expect(spy).toHaveBeenCalled();
- expect(spy).toHaveBeenCalledTimes(number);
- expect(spy).toHaveBeenCalledWith(...arguments);

Jasmine también viene con funciones que se pueden ejecutar antes de realizar un test, o después:

- **beforeAll**: Se ejecuta **antes** de pasar **todos** los tests de una suite.
- **afterAll**: Se ejecuta **después** de pasar **todos** los tests de una suite.
- **beforeEach**: Se ejecuta **antes** de cada test de una suite.
- **afterEach**: Se ejecuta **después** de cada test de una suite.

Por ejemplo:

```
describe('Hello world', () => {  
  
  let expected = "";  
  
  beforeEach(() => {
```

```

    expected = "Hello World";
  });

  afterEach(() => {
    expected = "";
  });

  it('says hello', () => {
    expect(helloWorld())
      .toEqual(expected);
  });
});

```

Antes de ejecutar el test definido mediante la función **it** se llama a la función **beforeEach** la cual cambia el valor de la variable `expected`, haciendo que el test pase.

Tests unitarios con Angular

Si has creado el proyecto y los componentes usando Angular cli, te habrás dado cuenta de que al generar un componente, también se crea un archivo **.spec.ts**, y eso es porque Angular cli se encarga por nosotros de generar un archivo para testear cada uno de los componentes. Además mete en el archivo el código necesario para empezar a probar y testear los componentes. Por ejemplo, el archivo **notes.component.spec.ts** que se creó cuando generé un componente para crear y mostrar notas tiene esta pinta:

```

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

import { NotesComponent } from './notes.component';

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

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ NotesComponent ]
    })
    .compileComponents();
  }));

```

```

beforeEach(() => {
  fixture = TestBed.createComponent(NotesComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

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

```

Lo primero que hace es crear una suite de tests para el componente con el método **describe**. Tras crear la suite crea dos variables que va a necesitar para testear los componentes, el propio componente, que lo mete en la variable **component** y una variable fixture de tipo ComponentFixture del componente, la cual sirve para tener el componente pero añadiendo más información para que sea más fácil de testear.

A continuación llama al método `beforeEach` con una función asíncrona (sirve para asegurar que se termina de ejecutarla función asíncrona antes de pasar un test) para crear todas las dependencias del componente, en ese caso, el componente en sí. Si usáramos en el componente un servicio, habría que incluirlo también, creando una sección llamada providers (como en el `app.module.ts`).

Después vuelve a llamar a la función **beforeEach**, esta vez, sin ser asíncrona. Crea una instancia fixture del componente usando `TestBed`, el cual se encargará de inyectar las dependencias definidas anteriormente mediante **configureTestingModule**. Para sacar el componente en sí del fixture usa **componentInstance**.

Por último crea un test para comprobar que el componente se crea correctamente, para ello, llama a la función **expect** y espera que se cree bien y tenga error mediante **toBeTruthy()**.

Para correr los tests y ver los resultados con Angular cli el comando es:

```
ng test
```

Testeando clases en Angular

Imaginemos que tenemos un servicio inyectado a un componente que queremos testear. Podemos usar varias técnicas para testear el servicio:

Usando el servicio real

```
import {LoginComponent} from './login.component';
import {AuthService} from './auth.service';

describe('Login component', () => {

  let component: LoginComponent;
  let service: AuthService;

  beforeEach(() => {
    service = new AuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => {
    localStorage.removeItem('token');
    service = null;
    component = null;
  });

  it('canLogin returns true when the user is authenticated', () => {
    localStorage.setItem('token', '12345');
    expect(component.isLogged()).toBeTruthy();
  });

});
```

En este caso, a diferencia de la estructura que crea Angular cli, no estoy usando TestBed, porque por el momento no me hace falta. Simplemente creo el componente y el servicio y paso el servicio como parámetro al componente para que se inyecte mediante inyección de dependencias. Cuando hago el test, simplemente llamo al método del componente y hago la comprobación.

Esta técnica puede venir bien para aplicaciones pequeñas, pero si el componente necesita muchas dependencias puede llegar a ser muy tedioso andar creando todos los servicios. Además esto no favorece la encapsulación porque estamos creando servicios y no

estamos aislando el testeo del componente.

Además de esta forma, tenemos que meter a mano en el localStorage un valor para que el authService funcione y devuelva **true**.

Creando un servicio virtual (mockeando)

```
import {LoginComponent} from './login.component';

class MockAuthService {
  authenticated = false;

  isAuthenticated() {
    return this.authenticated;
  }
}

describe('Login component', () => {

  let component: LoginComponent;
  let service: MockAuthService;

  beforeEach(() => {
    service = new MockAuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => {
    service = null;
    component = null;
  });

  it('canLogin returns true when the user is authenticated', () => {
    service.authenticated = true;
    expect(component.isLogged()).toBeTruthy();
  });
});
```

```
});
```

Esta vez, en lugar de usar el authService real, creamos nuestra propia clase **MockAuthService** dentro del propio test, la cual tendrá un método con el mismo nombre que el servicio real, pero en su lugar devuelve el valor directamente.

Como hacíamos antes, creamos el componente y le pasamos el servicio, en este caso, el servicio virtual que hemos creado. Usando este método no tenemos que usar el localStorage, de esta forma, solo testeamos el componente en sí y no tenemos que depender de lo que haga el servicio internamente.

Si aún así crear el servicio virtual resulta costoso, siempre podemos extender del servicio real, sobrescribiendo los métodos que nos interesen:

```
class MockAuthService extends AuthService {  
  authenticated = false;  
  
  isAuthenticated() {  
    return this.authenticated;  
  }  
}
```

También podemos sobrescribir la inyección de dependencias con nuevas clases, por ejemplo:

```
TestBed.overrideComponent(  
  LoginComponent,  
  {set: {providers: [{provide: AuthService, useClass: MockAuthService}]}}  
);
```

Mediante del uso de spy de Jasmine

Jasmine también ofrece la posibilidad de coger una clase y devolver directamente lo que nos interese sin tener que ejecutar internamente sus métodos:

```
import {LoginComponent} from './login.component';  
import {AuthService} from './auth.service';  
  
describe('Component: Login', () => {  
  
  let component: LoginComponent;  
  let service: AuthService;
```

```

let spy: any;

beforeEach(() => {
  service = new AuthService();
  component = new LoginComponent(service);
});

afterEach(() => {
  service = null;
  component = null;
});

it('canLogin returns true when the user is authenticated', () => {
  spy = spyOn(service, 'isAuthenticated').and.returnValue(true);
  expect(component.isLogged()).toBeTruthy();
});
});

```

Como ves, con la función `spyOn` de Jasmine podemos hacer que el servicio devuelva directamente `true` en la llamada a el nombre de función que le pasamos como parámetro al `spy`.

Testeando llamadas asíncronas

Si por ejemplo tenemos un test que testea un método asíncrono del componente o del servicio (una llamada a una API por ejemplo), podemos hacer lo siguiente:

```

it('Should get the data', fakeAsync(() => {
  fixture.componentInstance.getData();
  tick();
  fixture.detectChanges();
  expect(component.data).toEqual('new data');
}));

```

Angular proporciona el método **`fakeAsync`** para realizar llamadas asíncronas, dejándonos acceso a la llamada a **`tick()`** el cual simula el paso del tiempo para esperar a que la llamada asíncrona se realice.

Accediendo a la vista

Para acceder a los elementos html de la vista de un componente, podemos usar su fixture:

```
describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let submitButton: DebugElement;

  beforeEach(() => {

    TestBed.configureTestingModule({
      declarations: [LoginComponent]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    submitButton = fixture.debugElement.query(By.css('button_submit'));

  });
});
```

En este caso, accedemos al botón html de la vista del componente de login mediante el debugElement del fixture, el cual nos da acceso a hacer queries de elementos html de la vista.

Como en javascript podemos acceder o cambiar las propiedades de estos elementos:

```
submitButton.innerHTML = 'Testing the button';
```

El TestBed del componente nos proporciona una manera de provocar que la vista de actualice con la nueva información en caso de que hayamos cambiado algo en el componente:

`fixture.detectChanges()`

Testing de llamadas http

Para testear las llamadas HTTP podemos hacer dos tests, uno para comprobar que la petición se ha realizado correctamente y otro test para verificar que la información que llega de la API es correcta. Para lo segundo podemos usar la clase `MockBackend` de Angular, que es capaz de simular un backend con información creada por nosotros, para que cuando el servicio realice la llamada HTTP en realidad llame al `MockBackend` para que no tenga ni que hacer la llamada real y podamos comprobar la información que llega al servicio.

No me voy a meter mucho más en este tema, porque sinceramente es complejo de entender. Si aún así tienes dudas y quieres aprender como testear servicios con llamadas http te dejo esta lista de artículos (en inglés):

- <https://angular-2-training-book.rangle.io/handout/testing/services/http.html>
- <https://medium.com/spektrakel-blog/angular-testing-snippets-httpclient-d1dc2f035eb8>

Algunos consejos para terminar

Antes de acabar me gustaría darte unos pequeños consejos que seguramente te resulten útiles:

- Intentar tener más o menos clara la estructura que va a tener el proyecto, o por lo menos decide hacia donde va a ir. También es importante que decidas si vas a usar un gestor de estado tipo NgRx o Ngxs. Es importante para no tener que refactorizar más adelante todo el proyecto.
- Intentar modularizar y crear componentes reusables. En todo lenguaje de programación es muy importante reusar lo máximo posible el código para agilizar y para no tener que cambiar el código cada vez que necesites un cambio. Piensa en los componentes para que sean aislados y genéricos para favorecer la reusabilidad.
- Piensa en utilizar lo que ofrece Angular y antes de hacer nada mira si ya existe una solución dentro del ecosistema de Angular. Por ejemplo si necesitas cambiar un estilo CSS dinámicamente no uses el código Javascript como hasta ahora, utiliza lo que ofrece Angular para cambiar estilos.
- No descargues librerías si no son completamente necesarias. Por ejemplo hace tiempo en un proyecto necesitaba un componente de tablas. Estube mirando la opción de añadir una librería de componentes de tablas, pero decidí implementarme las mías propias.

Con esto consigues conocer a la perfección tu proyecto y tener un control total de lo que estás haciendo. Si necesitas un cambio en una librería lo tienes que proponer a su creador y no siempre te hacen caso.

- Intenta utilizar los principios de Optimistic UI. Optimistic UI consiste en una manera de mostrar la información sin esperar respuesta del servidor. Por ejemplo imagina que estás desarrollando una app de chat en tiempo real. Si cada vez que envías un mensaje tienes que esperar por la respuesta del servidor, la interfaz de usuario no resultará inmediata.

Una forma de resolver esto para este ejemplo sería, pintar directamente el mensaje en pantalla como si se hubiera enviado correctamente. Si la respuesta que llega del servidor es errónea entonces muestras el error. Esto consigue que de la apariencia al usuario de que la aplicación funciona rapidísima.

- Si la aplicación es grande, intentar aprovechar el potencial de typescript en cuanto a crear interfaces y modelos para los datos, eso evitará problemas a la larga. Además

tienes a tu disposición los tipos para las variables, lo que va a facilitar mucho su depuración

- Si puedes usa algún precompilador de código css como scss, eso agilizará mucho la creación de estilos.
- Crea tu propia librería de componentes. Esto es algo que empecé a usar hace poco. Te creas una librería con los componentes que quieras y los subes a un repositorio. Cuando tengas que empezar un proyecto nuevo de Angular, siempre puedes bajarte esa librería para tener componentes ya creados.

Conclusiones

Eso es todo, muchas gracias por haber dedicado tiempo a leer este libro. Espero que te haya gustado el libro, o que al menos hayas aprendido algún concepto que antes no sabías. He intentado proyectar en el libro todos mis conocimientos sobre Angular explicados de forma sencilla.

Angular no termina aquí, todavía tiene más cosas, es un framework muy grande que ofrece todo tipo de funcionalidades. Además según pasa el tiempo sacan nuevas versiones con más funcionalidad todavía. Es por eso que te animo a que sigas aprendiendo. En el mundo de la programación es vital estar siempre en continuo aprendizaje y mejora.

Con todo lo que has aprendido hasta ahora deberías ser capaz de crear aplicaciones web con Angular completas. Te animo que sigas descubriendo y a que trates de enseñar a la gente el potencial de este framework.

Como he dicho antes, muchas gracias por confiar en mí. Nos vemos en la próxima.

— Diego López García (@CodingPotions). Versión 1.1

Copyright (c) 2019 Coding Potions

Queda prohibida la venta y distribución de este libro.