

# Angular

Bootcamp Frontend Developer

# ANGULAR



Es un framework Javascript desarrollado y mantenido por Google y está basado en Typescript (un superset de JavaScript). Angular es el sucesor completamente reescrito e incompatible de AngularJS.

Muy importante: Angular JS != Angular

Primera Release: Octubre de 2010

Angular es usado por empresas como Google, Wix, weather.com, Forbes...

## Ventajas de usar Angular:

- Modularidad
- Inyección de dependencias
- Enrutamiento
- Validaciones
- Typescript (tipo C#, Java...)
- Separación clara entre HTML y Typescript

## Desventajas:

- Curva de aprendizaje
- Poco flexible

# ANGULAR



Angular es un framework basado en el patrón de diseño MVC o Modelo Vista Controlador

## MVC:

Una manera de diseñar software donde se requiere el uso de interfaces de usuario.

Separación de código en tres capas diferentes: Modelos, Vistas y Controladores.

Evitar Código Spaghetti

**Modelos:** Capa de Datos. Acceso y actualización de la información necesaria para nuestra aplicación

**Vistas:** Capa de visualización. Contiene la lógica que genera la interfaz de usuario.

**Controladores:** Capa de lógica. Enlace entre vista y modelo, contiene las acciones que se solicitan en la aplicación.

# ANGULAR



Como todos los frameworks front-end modernos se basa en la idea de separar la capa front de la capa back-end conectando ambos mediante servicios REST con peticiones Ajax.

Ventajas de una arquitectura orientada a servicios:

**ESCALABILIDAD:** El backend tiene menos trabajo y sólo procesa la lógica del negocio.

**INTEROPERABILIDAD:** Fácil integración con muchas plataformas

**DESARROLLO PARALELO:** Es posible tener un equipo sólo para el desarrollo front y otro para el backend

Este estilo de arquitectura impulsó la Web 2.0. Esta es una de las muchas técnicas, movimientos y culturas que ofrecen una nueva perspectiva para el lado del cliente.

Service-Oriented Front-End Architecture == Single Page Application (SPA)

# ANGULAR



Como ya hemos adelantado Angular se basa en concepto de SPA (Single Page Application)

- Una SPA no es una aplicación con una sola página HTML, si no una aplicación contenida completamente en el navegador que no necesita más peticiones de páginas al servidor
- Normalmente una SPA hace peticiones únicamente de los datos que se mostrarán dentro de las páginas, solicitando los datos a servicios REST+JSON
- Son mucho más rápidas al eliminar la carga de HTML, CSS y JS con cada petición (y las cabeceras consiguientes)
- Permite crear aplicaciones OFFLINE

# TYPESCRIPT



¿Qué es TypeScript? ¿Por qué Angular utiliza TypeScript en lugar de JavaScript?

- TypeScript es en realidad un superconjunto de JavaScript. Se podría decir que TypeScript es a JavaScript como SCSS es a CSS. El código JavaScript, es código Typescript valido, porque Typescript es un superconjunto de JavaScript.
- Hace que el código sea más fácil de leer y de entender.
- Proporciona “tipado” sobre JavaScript, aunque sea sólo en tiempo de desarrollo. Esto hace que cometamos menos errores a la hora de programar.
- Typescript soporta los nuevos estándares ECMAScript y los compila a versiones más antiguas. Esto significa que podemos utilizar las nuevas características de las nuevas versiones del estándar en navegadores que aún no las soportan.

# TYPESCRIPT: tipado

TypeScript tiene tipado estático: las variables llevan asociado el tipo de dato y no se puede cambiar.

Tenemos básicamente los mismos tipos que tenemos en JavaScript:

- string
- number
- boolean
- object

Y algunos más:

- any: Indica que esta variable puede ser de cualquier tipo
- void: void significa que esta función no devuelve un valor
- tipos “custom”: podemos crear nuestros propios tipos definiendo interfaces

```
let nombre: string = "Jorge";  
  
function miFuncion(param: string): number {  
    return 1  
}  
  
interface Circular {  
    numeroVueltas: number;  
    nombre: string;  
}  
  
let miDatos: Circular = {  
    numeroVueltas: 10,  
    nombre: "Vuelta Molona"  
}
```

# ANGULAR CLI



Es la herramienta de línea de comandos creada por el mismo equipo de Angular, requiere de NodeJs y se instala mediante npm:

**npm install -g @angular/cli**

Permite crear el esqueleto de una aplicación angular:

**ng new my-app-nueva**

Además incorpora una serie de herramientas útiles para el desarrollo como son un servidor para servir el proyecto por http, un sistema de live-reload, compilado del SASS y del TypeScript.

Para “levantar” el proyecto:

**ng serve**

Para generar un componente nuevo (recuerda parar el serve con ctrl + C):

**ng generate component header**

# ANGULAR: Componentes



Angular está pensado para desarrollar web “partiéndolas” en componentes.

Un componente es un bloque o porción de la aplicación

En Angular 2, TODO ES UN COMPONENTE

Los componentes contienen sus propios templates, estilos y lógica, de modo que se pueden reutilizar en diferentes contextos

La aplicación en sí misma es un componente, del que “cuelgan” el resto de componentes



# ANGULAR: Componentes



Un componente de Angular, tiene:

- Un selector: es el nombre del componente y lo usaremos para pintarlo
- Un template: será la ruta del HTML de nuestro componente
- En un array de ficheros de estilos (css, sass/scss)
- Imports de las clases / módulos necesarios
- Constructor de nuestro componente
- Los métodos que necesitemos

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

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

export class AppComponent {
  title = 'my-app-nueva-angular';

  constructor() {
    ...
  }
}
```

# ANGULAR: Comp. (Template)



Un template es un HTML que le indica a Angular cómo renderizar el componente. Puede contener bindeo de datos e incluir otros componentes.

Data binding:

Para renderizar un valor se usan las clásicas llaves  
{{ variable }}

Para bindear propiedades a un componente usamos  
corchetes:  
propiedad=[valor]

Para escuchar eventos que pueden ser emitidos desde el  
componente usamos paréntesis:  
(click)="manejarClick()"

```
<h1>Bienvenido a {{ tituloDeLaPagina }}!</h1>

<app-menu [links]="misEnlaces"></app-menu>

<button (click)="userWantToSend()">Enviar</button>
```

# ANGULAR: Comp. (Estilos)



Encapsulation:

Permite a los estilos ser aplicados a un solo componente

Hay tres tipos de encapsulación en Angular 2:

**ViewEncapsulation.None.** Los estilos son globales y accesibles por toda la aplicación. Es lo más parecido a usar CSS de forma habitual

**ViewEncapsulation.Emulated.** Los estilos son “copiados” añadiendo atributos a los elementos dentro del template del componente. Este es el comportamiento por defecto de Angular 2

**ViewEncapsulation.Native.** Usa el Shadow DOM para insertar los estilos

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
  encapsulation: ViewEncapsulation.None
})

export class AppComponent {
  title = 'my-app-nueva-angular';

  constructor() {

  }
}
```

# ANGULAR: Comp. (@Input)



La propiedad `@Input` de los componentes nos permite pasárselos parámetros, como si de una función se tratara. Esto nos permite hacer los componentes “configurables”. Pasaremos los datos desde un componente padre a un componente hijo, de esta manera podemos por ejemplo repetir un componente botón pero con títulos diferentes.

## `@Input:`

- Se define dentro del componente de la siguiente forma:

```
@Input() nombreVariable: string = "valor por defecto";
```

- Se bindea un dato desde el padre así:

```
<component [nombreInput]="valorInput"></component>
```

# ANGULAR: Comp. (@Input)



Componente hijo (fichero.ts):

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

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

export class ButtonComponent {
  @Input() title: string = null;
  constructor() { }
}
```

Componente padre (fichero.html):

```
<app-button [title]="'Aceptar'"></app-button>
<app-button [title]="'Cancelar'"></app-button>
```

Componente hijo (fichero.ts):

```
<button class="boton">{{ title }}</button>
```

# ANGULAR: Comp. (@Output)



La propiedad `@Output` nos permite que los componentes emitan información hacia afuera, de manera que puedan comunicarse con “el exterior”. Los componentes hijos emitirán un valor y este valor será recogido por una función del componente padre.

## `@Output`:

- Se define en el componente con el decorador `@Output`:

```
@Output() nombreEmision = new EventEmitter();
```

- Bindeado en el componente padre de la siguiente forma:

```
<component (nombreEmision)=“nombreFuncion()”></component>
```

- Para emitir un valor se debe ejecutar:

```
this.nombreEmision.next(valorEmitido);
```

# ANGULAR: Comp. (@Output)



Componente hijo (fichero .ts):

```
import { Component, Input, Output, EventEmitter } from '@angular/core';

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

export class ButtonComponent {
  @Input() title: string = null;
  @Output() userClickedMoment: EventEmitter<Date> = new EventEmitter();

  public emitUserClick() {
    let valueToEmit = new Date();
    this.userClickedMoment.next(valueToEmit);
  }
}
```

Componente hijo (fichero .html):

```
<button class="boton" (click)="emitUserClick()">{{ title }}</button>
```

Componente padre (fichero .html):

```
<app-button
  (userClickedMoment)="userClikAccept($event)"
  [title]="'Aceptar'">
</app-button>
```

Componente padre (fichero .ts):

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

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

export class SearchFormComponent {

  public userClikAccept(moment: Date) {
    console.log("Ha pulsado aceptar:");
    console.log(moment);
  }
}
```

# Componentes Tontos vs Listos



## Componentes listos o “Containers”:

- Están conectados a servicios
- Saben como cargar sus datos y cómo persistir los cambios

## Componentes tontos o “Componentes de presentación”:

- Están definidos completamente por sus bindeos / propiedades
- Todos sus datos entran como inputs y todos los cambios salen como output

Crea los menos componentes listos posibles y los más componentes tontos posibles

En nuestras apps debemos tener de ambos tipos. Los tontos nos permitirán pintar datos, por ejemplo una tabla, un acordeón... y los componentes listos nos permitirán recuperar los datos de una API.

# Fuentes en Angular



No olvidemos que Angular es un framework de desarrollo front basado en JavaScript / TypeScript. La mayor parte de las cosas que debemos aprender de Angular aplican a nivel de lógica o de renderizado de la vista, pero generalmente no de estilos. Los estilos quedan delegados a SASS / SCSS / CSS... etc según hayamos configurado nuestro proyecto.

En una aplicación basada en componentes, encontraremos hojas de estilo asociadas a cada componente, por ejemplo:

- footer.component.scss
- button.component.scss...etc

Y hojas de estilo globales:

- styles.scss
- app.components.scss

Generalmente las fuentes deben aplicar de manera global por lo que deberemos definir nuestras fuentes en estos últimos.

# Fuentes en Angular



Podríamos “copiar” la definición de la fuente directamente en nuestro styles.scss y también ahí indicar quién la usará, pero si queremos ser organizados deberíamos crear un fichero de fuentes separado e importarlo en nuestro styles.scss. Por ejemplo:

Fichero fonts.scss:

```
// Importado de fuentes
@import './roboto.scss';
@import './helvetica.scss';
@import './coming-soon.scss';

// Definición de uso
body {
  font-family: 'Roboto', sans-serif;
}

h1, h2, h3, h4, h5, h6 {
  font-family: 'Coming Soon', cursive;
}

a, ol, ul {
  font-family: 'Helvetica', cursive;
}
```

Fichero roboto.scss:

```
@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 400;
  font-display: swap;
  src: local('Roboto'), local('Roboto-Regular'),
       url(https://fonts.gstatic.com/s/robot...
  unicode-range: U+0460-052F, U+1C80-1C88...
}

@font-face {
  font-family: 'Roboto';
  font-style: normal;
  font-weight: 400;
  font-display: swap;
}
// ETC...
```

Fichero styles.scss:

```
// Importado de fuentes
@import './fonts/fonts.scss';
```

# Directivas Estructurales



Las directivas estructurales nos permiten modificar el HTML de forma dinámica dependiendo del valor de una variable. Esto nos servirá para añadir elementos, borrarlos o repetirlos, como decíamos, dependiendo de los valores de las variables JavaScript de nuestro componente.

Son fáciles de reconocer, las directivas estructurales empiezan por asterisco (\*)



# Condicionales con \*ngIf



Componente, fichero .ts

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

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss'],
})
export class HeaderComponent {
  public title = "Hola soy el título";
  constructor() {}

}
```

Componente, fichero .html

```
<div class="header">
  <!-- Podemos comprobar si está definido... -->
  <div class="header__title" *ngIf="title">{{ title }}</div>
  <div class="header__title" *ngIf="!title">Lo siento, no hay título</div>

  <!-- O cualquier otra condición... -->
  <div class="header__title" *ngIf="title.length < 10">{{ title }}</div>
  <div class="header__title" *ngIf="title.length >= 10">Título muy largo</div>
</div>
```

# Bucles con \*ngFor



Componente, fichero .ts

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

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

export class FooterComponent {
  public links: LinkItem[] = [
    { href:"http://google.es", title: "Google" },
    { href:"http://marca.com", title: "Marca" },
    { href:"http://as.com", title: "As" }
  ];

  export interface LinkItem {
    href: string;
    title: string;
  }
}
```

Componente, fichero .html

```
<div class="footer">
  <div class="footer__link" *ngFor="let link of links">
    <a href="{{ link.href }}>{{ link.title }}</a>
  </div>
</div>
```

# Clases dinámicas (ngClass)



En ciertas ocasiones nos interesa que un elemento tenga una clase dependiendo del valor de una variable. Por ejemplo: en un listado de tareas si queremos pintar en verde aquellas tareas que están realizadas. Para este tipo de tareas Angular nos provee ngClass. Supongamos este app.component.ts:

```
export class AppComponent {
  public tasks: Task[] = [
    { title: "Bajar la basura", done: true },
    { title: "Sacar al perro", done: false },
    { title: "Preparar el tupper", done: false },
    { title: "Llamar a los abuelos", done: true },
  ];
  constructor() { }
}

export interface Task {
  title: string;
  done: boolean;
}
```

# Clases dinámicas (ngClass)



Podemos aprovechar la variable “done” de nuestros objetos Task para pintar una clase “task\_\_name--done” que muestre en verde aquellas tareas realizadas:

```
<div class="task" *ngFor="let task of tasks">
  <div class="task__name" [ngClass]="{ 'task__name--done': task.done == true }">
    {{ task.title }}
  </div>
</div>
```

Y de esta manera por CSS indicar que las tareas saldrán rojas, excepto las realizadas que saldrán verde:

```
.task{
  &__name {
    color: red;

    &--done {
      color: green;
    }
  }
}
```

Resultado:

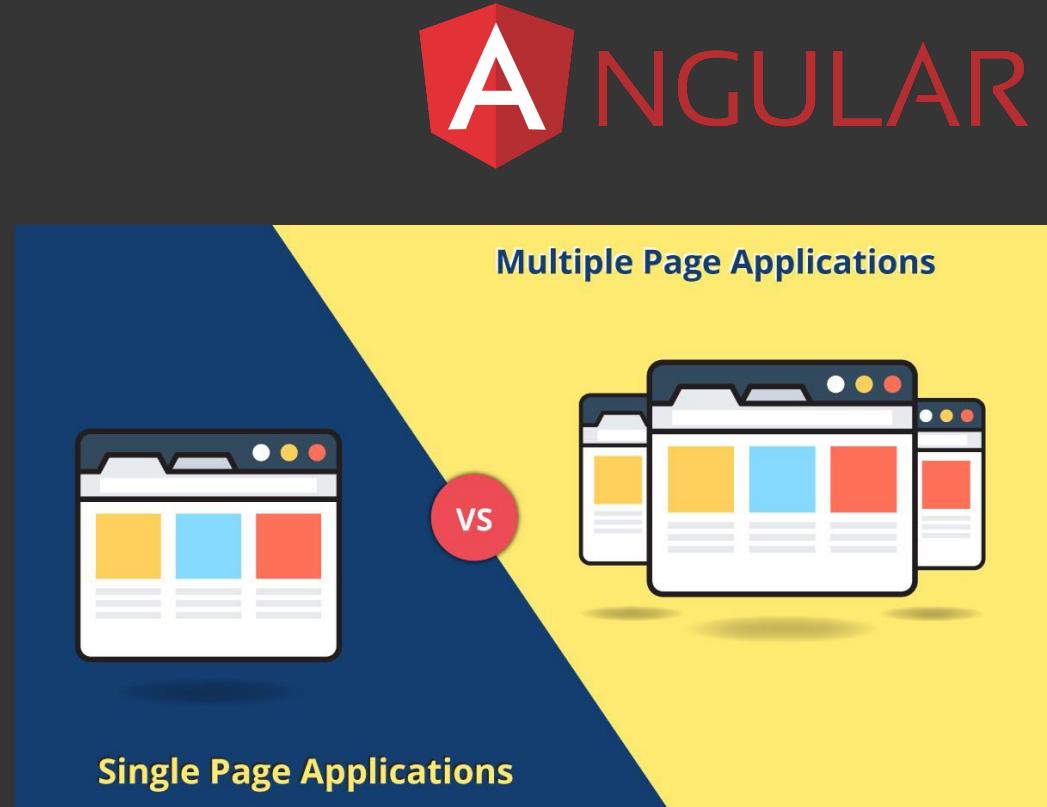
Bajar la basura  
Sacar al perro  
Preparar el tupper  
Llamar a los abuelos

# Angular Routing

La manera clásica de trabajar es realizar diversos documentos html e ir enlazando entre ellos.

Con la aparición de los nuevos estándares de HTML5, se abrió la posibilidad de realizar aplicaciones con otro enfoque, el enfoque SPA (Single Page Application).

Dentro de una SPA el usuario también navega, salvo que no carga documentos completos, sino que renderiza parte de la vista. Angular trae un módulo que implementa todo esto (Routing).



Las rutas le indican a Angular qué vistas mostrar y cuando, redirigen al usuario a otro componente sin recargar la página o llamar al backend / API, son activadas (triggered) por alguna interacción con la interfaz.

# Angular Routing



El “Router” es un Servicio opcional, se implementa a través del módulo opcional “RouterModule” que NO forma parte del core (@angular/core) de Angular, debe ser instalado con npm (@angular/router), aunque actualmente el CLI de Angular te pregunta si deseas instalarlo durante la creación del proyecto.

Importación (app.module.ts):

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

# Angular Routing



Después mediante el módulo de routing que acabamos de importar, definiremos cada una de las rutas y los componentes que cargarán. Definición de rutas (app-routing.module.ts):

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AboutUsPageComponent } from './about-us-page/about-us-page.component';
import { WhereWeArePageComponent } from './where-we-are-page/where-we-are-page.component';
import { HomePageComponent } from './home-page/home-page.component';

const routes: Routes = [
  { path: 'home', component: HomePageComponent },
  { path: 'about-us', component: AboutUsPageComponent },
  { path: 'where-we-are', component: WhereWeArePageComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
];

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

# Angular Routing



Ahora necesitamos dos cosas, por un lado crear enlaces para navegar y por otro indicar dónde se van a pintar esos componentes:

Fichero app.component.html:

```
<!-- Links para cargar las rutas de Angular -->
<a routerLink="/home" routerLinkActive="active">Home</a>
<a routerLink="/about-us" routerLinkActive="active">About Us</a>
<a routerLink="/where-we-are" routerLinkActive="active">Where We Are</a>

<!-- Aquí se pintarán los componentes definidos en el router -->
<router-outlet></router-outlet>
```

# Angular Routing: Params



También podemos pasar parámetros en las rutas. Esto puede ser muy útil por ejemplo si tenemos una página con un detalle, por ejemplo una película, un usuario, una ciudad... etc. lo correcto en esos casos es que cada uno tenga su propia ruta:

- /movie/14123
- /user/mfernandez
- /city/madrid

Para ello, angular nos provee la posibilidad de declarar rutas con parámetros (app-routing.module.ts):

```
const routes: Routes = [
  { path: 'home', component: HomePageComponent },
  { path: 'about-us', component: AboutUsPageComponent },
  { path: 'where-we-are', component: WhereWeArePageComponent },
  { path: 'user/:id', component: UserDetailComponent }, // RUTA CON PARAMS
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

# Angular Routing: Params



Y por supuesto, nos provee también una forma de recuperar ese parámetro desde el componente (`user-detail.component.ts`):

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

@Component({
  selector: 'app-user-detail',
  templateUrl: './user-detail.component.html',
  styleUrls: ['./user-detail.component.scss']
})
export class UserDetailComponent {
  public userId: string = null;

  constructor(private activatedRoute: ActivatedRoute) {
    this.activatedRoute.params.subscribe((params) => {
      if (params && params.id) {
        this.userId = params.id;
      }
    });
  }
}
```

# Angular Routing: Params



Y de crear enlaces con params:

```
<!-- Links para cargar las rutas de Angular -->
<a routerLink="/home" routerLinkActive="active">Home</a>
<a routerLink="/about-us" routerLinkActive="active">About Us</a>
<a routerLink="/where-we-are" routerLinkActive="active">Where We Are</a>
<a routerLink="/bla-bla-bla" routerLinkActive="active">Bla bla bla</a>

<!-- Links con params -->
<a [routerLink]=["/user", 'fjlinde"]>Ver detalles de fjlinde</a>
<a [routerLink]=["/user", 'dbarredo"]>Ver detalles de dbarredo</a>
```

# Angular Routing: Nav desde JS



Hasta ahora hemos navegado creando enlaces en el HTML, pero en muchas ocasiones es necesario que la navegación se dispare desde el JS no desde el HTML. Aquí un ejemplo :

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  public users: string[] = ["pfnieva", "iabajo", "acuenca"];

  constructor(private router: Router) {}

  public seeLastUser() {
    let lastUser = this.users[this.users.length-1];
    this.router.navigate(["user", lastUser]);
  }
}
```

Y el correspondiente template

app.component.html

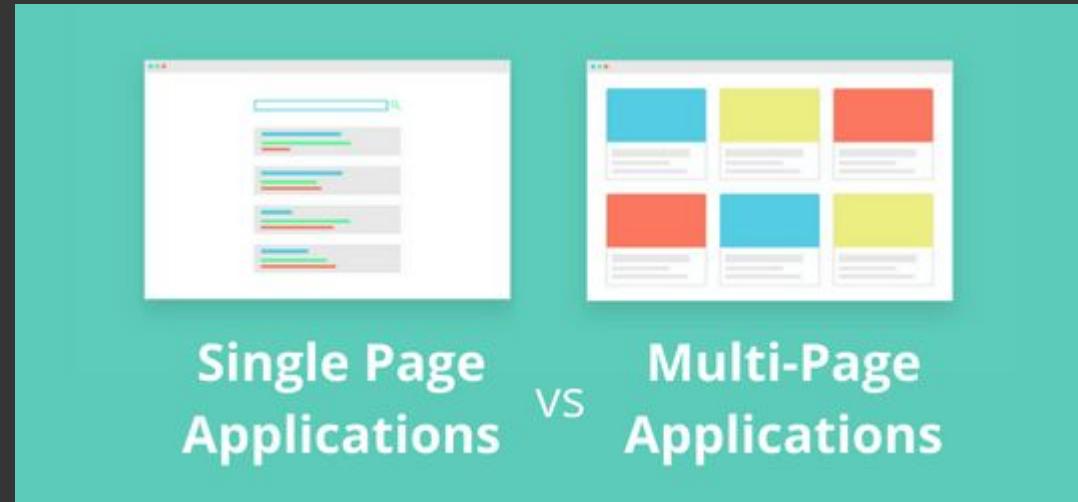
```
<button (click)="seeLastUser()">Ver el último user</button>
```

# Peticiones AJAX en Angular



Como ya hemos mencionado anteriormente las peticiones AJAX nos permiten consumir contenido (generalmente en formato JSON) sin tener que recargar la página. Este es uno de los pilares principales de las aplicaciones SPA (Single Page Application) junto con el enrutamiento.

Hasta ahora hemos visto cómo hacer peticiones mediante XMLHttpRequest y Fetch, ambos son métodos de JavaScript nativo (también llamado Vanilla). Cuando trabajamos con Angular las peticiones debemos hacerlas haciendo uso de sus propios métodos y objetos, que vamos a ver a continuación.



# Peticiones AJAX en Angular



Para poder hacer peticiones en Angular, en primer lugar debemos importar el módulo `HttpClientModule` dentro de nuestro `AppModule`.

`app.module.ts`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';

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

# Peticiones AJAX en Angular



Ahora ya podemos hacer uso del HttpClient para realizar peticiones. Eso sí, no sería correcto que hagamos las peticiones directamente desde los componentes, por este motivo crearemos servicios/provider. Los servicios de Angular nos permiten crear una capa de acceso a datos común para todos los componentes. De esta manera tendremos nuestro código organizado y también podremos compartir datos entre componentes guardandolos en esos servicios. Para crear un servicio ejecutaremos:

```
ng generate service NombreDelServicio
```

Esto nos generará el fichero nombre-del-servicio.service.ts:

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

@Injectable({
  providedIn: 'root'
})
export class CharactersService {

  constructor() { }

}
```

# Peticiones AJAX en Angular



Dentro de nuestro servicio recién generado podemos importar el HttpClient, el cual nos permitirá realizar las peticiones. Y debemos crear métodos que devuelvan nuestros datos. Teniendo en cuenta que las peticiones son asíncronas, debemos usar Observables. Los observables (implementados por la librería RxJS) se parecen mucho a las promesas o a los callbacks. Proveen un mecanismo para poder trabajar con la asíncronía:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class NombreDelServicioService {
  private apiUrl = "https://api.got.show/api/show/characters";

  constructor(private httpClient: HttpClient) {}

  getData(): Observable<any> {
    let observable: Observable<any> = this.httpClient.get(this.apiUrl);
    return observable;
  }
}
```

# Peticiones AJAX en Angular



Y desde nuestros componentes debemos importar el servicio recién creado, para después usar esos métodos que nos devolverán un observable, para suscribirnos al evento de llegada de la información. De esta manera recuperaremos los datos:

```
import { Component, OnInit } from '@angular/core';
import { NombreDelServicioService } from '../nombre-del-servicio.service';

@Component({
  selector: 'app-characters-list',
  templateUrl: './characters-list.component.html',
  styleUrls: ['./characters-list.component.scss']
})
export class CharactersListComponent implements OnInit {
  public characters = [];

  constructor(private nombreDelServicioService: NombreDelServicioService) {}

  ngOnInit() {
    this.nombreDelServicioService.getData().subscribe(
      (data) => {
        this.characters = data;
      }
    );
  }
}
```