

<b>ANGULAR</b>	<b>2</b>
Instalación de Angular	2
Instalación de Visual Studio Code	2
Angular CLI	2
TypeScript	3
Variables de funciones	3
let vs const	4
Funciones	4
Arrow Function	4
Decorators @Input() and @Output()	5
Property Binding	6
One-way in	6
Pasar variables entre componentes	6
Pasar objetos entre componentes	6
Binding Targets	7
Depuración	8
Instalar Paquetes o Frameworks de CSS (Ej: Bootstrap)	9
Instalar Librerías de Javascript (Ej: lodash)	9
Structural Directives	10
Attribute Directives	10
Services	11
Decorator @Injectable()	12
Forms (Template-driven)	13
FormsModule	13
Template reference variable	13
ngForm Directive	14
ngModel Directive	14
Validators	15
Routing	16
Server-side vs Client-side	16
RouterModule	16
Redireccionamientos	18
Child Routes & Parámetros	18
ActivatedRoute	19
Router (Envío de Datos)	20
Router (Lectura de Datos)	20
Observables “subscribe()”	21
RxJS	21
Subject (Observable)	21

Gestión de memoria	22
Http Client	23
GET	23
POST	24
Custom Attribute Directives	25
Structural Directives NgFor (Avanzado)	26
Structural Directives NgSwitch	26
Angular Animations	28
Lista de funciones	29
Recursos para el diseño y creación de animaciones	30
Deployment	30
Recursos	31
Ejemplos	32
Comunicación entre 2 componentes	32
Uso de directivas estructurales y de atributos	32
Caso práctico 1	32
Caso práctico 1 pasado a Services	32
Caso práctico 1 con Formulario	32
Formulario de registro	32
Routers caso práctico 1	33
Routers caso práctico 2	33
RxJS caso práctico 1	33
Angular Role Game Project	33
Advanced directives and custom attribute directives	33
Simple Animation	33

# ANGULAR

## Instalación de Angular

1. Descargar NodeJS desde el enlace <https://nodejs.org/en/download/> el Windows Binary (.zip) de 64-bit. Y extraerlo en la carpeta que se desee.
2. Ejecutar el archivo "nodevars.bat" en el directorio descomprimido de NodeJS desde un terminal (CMD). Es posible que el directorio en el que estamos cambie, tendremos que volver al directorio de NodeJS.
3. Instalar el cliente de Angular (Angular CLI) usando la sentencia:  
**npm install -g @angular/cli**

## Instalación de Visual Studio Code

1. Descargar Visual Studio Code el archivo .zip de 64 bits desde el enlace <https://code.visualstudio.com/Download>. Y extraerlo en la carpeta que se desee.
2. Ejecutar el archivo **Code** dentro de la carpeta donde se ha extraído.
3. Una vez dentro del programa hay que instalar las siguientes extensiones:
  - a. vscode-icons
  - b. Angular Essentials

## Angular CLI

Commandos
<b>ng new my-first-project</b> <i>Crea un nuevo proyecto con nombre "my-first-project"</i>
<b>ng serve</b> <i>Arranca el servidor desde el directorio del proyecto que hayamos creado</i>
<b>ng g component "components/component-name"</b> <i>Desde el directorio del proyecto, crea el componente "component-name" en la carpeta "components", si no se especifica un nombre de carpeta, se crea automáticamente una nueva con el nombre del componente.</i>
<b>ng g service "services/service-name"</b>

*Desde el directorio del proyecto, crea el servicio “service-name” en la carpeta “services”, si no se especifica un nombre de carpeta, el servicio se crea en la carpeta raíz, o sea la carpeta “app”.*

#### **ng g directive “directives/directive-name”**

*Desde el directorio del proyecto, crea la directiva “directive-name” en la carpeta “directives”, si no se especifica un nombre de carpeta, la directiva se crea en la carpeta raíz, o sea la carpeta “app”.*

## TypeScript

### Variables de funciones

Resumen de tipos predefinidos del lenguaje:

KEYWORD	DESCRIPTION
number	It is used to represent both Integer as well as Floating-Point numbers
boolean	Represents true and false
string	It is used to represent a sequence of characters
void	Generally used on function return-types
null	It is used when an object does not have any value
undefined	Denotes value given to uninitialized variable
any	If variable is declared with any data-type then any type of value can be assigned to that variable

Ejemplos:

```
let a: null = null;
```

```
let a: any = null;
```

```
let b: number = 123;
```

```
let b: any =123;
```

```
let c: number = 123.456;
```

```
let c: any = 123.456;
```

```
let d: string = ‘Geeks’;
```

```
let d: any = ‘Geeks’;
```

```
let e: undefined = undefined;
```

```
let e: any = undefined;
```

```
let f: boolean = true;
```

```
let f: any = true;
```

```
let g: number = 0b111001; // Binary
```

```
let i: number = 0xadf0d; // Hexa-Decimal
```

## let vs const

Para definir variables podemos usar tanto la palabra clave **let** como **const**, la diferencia radica en que la primera nos permitirá modificar la variable y la segunda no.

## Funciones

Para definir funciones en TypeScript hay que se pone el nombre de la función seguido de los parámetros que le pasamos entre paréntesis, especificando el tipo de cada parámetro y por último el tipo de retorno de la función. En TypeScript no es obligatorio el tipo de los parámetros ni el tipo de retorno de la función y por tanto se pueden omitir. **No obstante es importante saber en cada momento con que tipo de datos trabajamos y que tipo de datos devolvemos.**

Función	Función sin definir tipos
<pre>mostrarTexto(texto: string): void {     console.log(texto); }</pre>	<pre>mostrarTexto(texto) {     console.log(texto); }</pre>

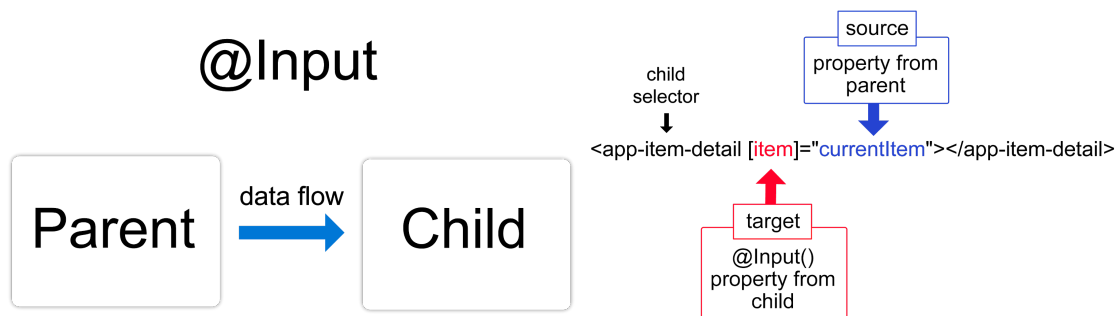
## Arrow Function

De la misma forma, una función puede escribirse de otra forma cuando es necesaria utilizarla en momentos puntuales. Y al igual que las funciones normales, se puede o no omitir los tipos de los parámetros y retorno.

Arrow function	Arrow function sin definir tipos
<pre>(variable: string): void =&gt; { console.log(variable); }</pre>	<pre>(variable) =&gt; { console.log(variable); }</pre>

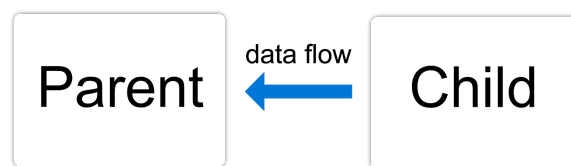
## Decorators @Input() and @Output()

El “decorator” @Input() lo utilizaremos siempre que necesitemos pasar información del componente padre al componente hijo. Y el “decorator” tendrá que estar en el componente hijo.

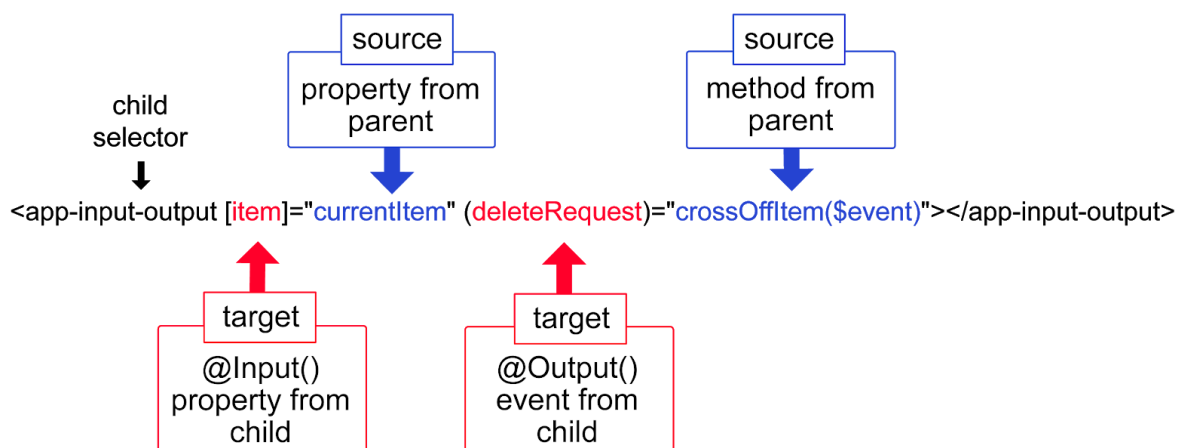


El “decorator” @Output() lo utilizaremos siempre que necesitemos pasar información del componente hijo al componente padre. Y el “decorator” tendrá que estar en el componente hijo.

## @Output



Si usamos los dos “decorators” juntos @Input() and @Output()



# Property Binding

<https://angular.io/guide/template-syntax#property-binding-property>

Los “Property Binding” se usan para dar un valor a las propiedades de los elementos, hay dos formas:

- Elementos Objetivo
- Decoradores @Input()

## One-way in

El “property binding” fluye en una sola dirección, el valor de la propiedad del elemento objetivo es el valor de la propiedad del componente.

```
<!-- Bind image src property to `itemImageUrl` component property -->
<img [src]="itemImageUrl">

<!-- Bind button disabled state to `isUnchanged` component property -->
<button [disabled]="isUnchanged">Disabled Button</button>
```

## Pasar variables entre componentes

En este caso desde el componente **app.component** le pasamos la propiedad **parentItem** al componente **item-detail.component**.

src/app/app.component.html

```
<app-item-detail [childItem]="parentItem"></app-item-detail>
```

src/app/item-detail/item-detail.component.ts

```
@Input() childItem: string;
```

src/app/app.component.ts

```
parentItem = 'lamp';
```

## Pasar objetos entre componentes

En este caso se le pasa un array de objetos de tipo **Item** al **list-item.component** desde el **app.component**.

src/app/app.component.html

```
<app-list-item [items]="currentItems"></app-list-item>
```

src/app/list-item.component.ts

```
@Input() items: Item[];
```

src/app/item.ts

```
export class Item {
```

```

    id: number;
    name: string;
  }

```

`src/app.component.ts`

```

currentItems = [{
  id: 21,
  name: 'phone'
}];

```

## Binding Targets

En el siguiente enlace podréis encontrar una tabla sumario de las diferentes posibilidades de binding.

<https://angular.io/guide/template-syntax#binding-targets>

Type	Target	Examples
Property	Element property	<p><code>src</code>, <code>hero</code>, and <a href="#">ngClass</a> in the following:</p> <pre> &lt;img [src]="heroImageUrl"&gt;  &lt;app-hero-detail [hero]="currentHero"&gt; &lt;/app-hero-detail&gt;  &lt;div [<a href="#">ngClass</a>]="{'special': isSpecial}"&gt; &lt;/div&gt; </pre>
	Component property	
	Directive property	
Event	Element event	<p><code>click</code>, <code>deleteRequest</code>, and <code>myClick</code> in the following:</p> <pre> &lt;button (click)="onSave()"&gt;Save&lt;/button&gt;  &lt;app-hero-detail (deleteRequest)="deleteHero()"&gt; &lt;/app-hero-detail&gt;  &lt;div (myClick)="clicked=\$event" clickable&gt;click me&lt;/div&gt; </pre>
	Component event	
	Directive event	
Two-way	Event and property	<pre> &lt;input [(<a href="#">ngModel</a>)]="name"&gt; </pre>



Attribute	Attribute (the exception)	<code>&lt;button [attr.aria-label]="help"&gt; help&lt;/button&gt;</code>
Class	class property	<code>&lt;div [class.special]="isSpecial"&gt;Special&lt;/div&gt;</code>
Style	style property	<code>&lt;button [style.color]="isSpecial ? 'red' : 'green'"&gt;</code>

[Ir al ejemplo "Comunicación entre 2 componentes"](#)

## Depuración

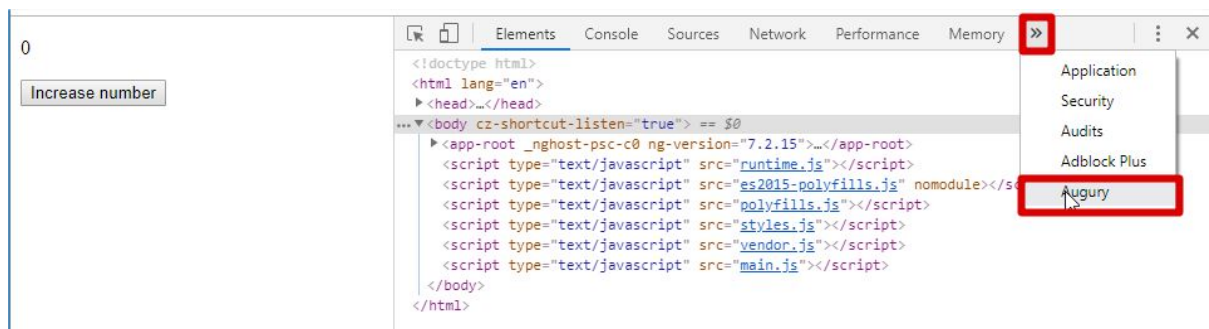
Para abrir las Chrome DevTools, desde Google Chrome → Apretar F12.

- Desde la pestaña de **Console** veremos los errores.
- Para depurar el código Typescript, **Sources** seleccionamos **webpack://** y dentro buscamos el directorio **app** allí estarán los archivos .ts de nuestro proyecto, donde encontraremos

Existe adicionalmente una extensión de Chrome la cual podemos descargar para que nos ayude con la parte del framework de Angular. Se llama **Augury** y la podemos descargar aquí:

<https://augury.rangle.io/>

Para verla, dentro de las Chrome DevTools podemos hacer como se muestra en la figura a continuación:



## Instalar Paquetes o Frameworks de CSS (Ej: Bootstrap)

Desde el directorio del proyecto en un terminal:

**npm install bootstrap --save**

Si se ha instalado aparecerá la dependencia en el archivo **package.json** de nuestro proyecto (es lo que hace el flag **--save**), de esta forma quien quiera usar nuestro proyecto con el comando **npm install** se instalarán todas las dependencias. Para usar el paquete instalado de Bootstrap tendremos que abrir el fichero **angular.json** y dentro de la llave **"styles"** añadir el fichero de estilos de Bootstrap, que podemos encontrar en la carpeta del proyecto **node\_modules**. Por defecto será:

```
"styles": [ "src/styles.css",  
            "node_modules/bootstrap/dist/css/bootstrap.min.css" ]
```

Para que funcionen todas las clases de bootstrap, necesitaremos al menos: **popper.js@^1.16.1** y **jquery@1.9.1** y añadir en el archivo **angular.json** las siguientes líneas y reiniciar el servidor.

```
"scripts": [ "node_modules/popper.js/dist/umd/popper.js",  
             "node_modules/jquery/jquery.js",  
             "node_modules/bootstrap/dist/js/bootstrap.bundle.js" ]
```

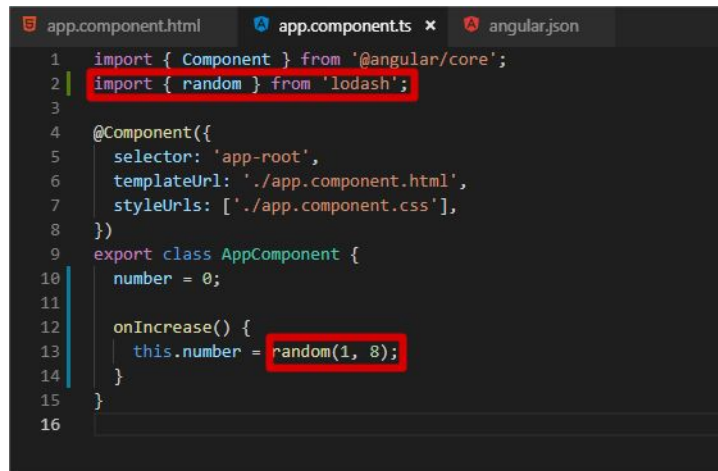
Es posible que se requieran dependencias de librerías de JavaScript, en ese caso ponerlas dentro de la llave **"scripts"**. Una vez añadido, paramos y volvemos a arrancar el servidor.

## Instalar Librerías de Javascript (Ej: lodash)

Para instalar librerías necesitaremos las siguientes instrucciones:

**npm install lodash --save**  
**npm install --save @types/lodash**

La primera instala las librerías en JS, y la siguiente las "traducciones" para TypeScript. Ejemplo de uso:



```
1 import { Component } from '@angular/core';
2 import { random } from 'lodash';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.css'],
8 })
9 export class AppComponent {
10   number = 0;
11
12   onIncrease() {
13     this.number = random(1, 8);
14   }
15 }
16
```

Documentación página oficial: <https://angular.io/guide/using-libraries>

## Structural Directives

Modifican o cambian la estructura del DOM de la página web.

### ngFor

En este caso creará tantos elementos del mismo tipo en el que está la directiva **\*ngFor** como elementos haya en el objeto **items**. Es una directiva estructural.

```
<ul class="list-group">
  <li class="list-group-item" *ngFor="let it of items">{{ it }}</li>
</ul>
```

### ngIf

Nos permite añadir o quitar algo del DOM en función de si la condición es cierta o falsa, en este caso si la variable **newItem** está vacía nos añadirá el párrafo con el texto que tenemos, en caso contrario lo eliminará. Es una directiva estructural.

```
<p *ngIf="newItem === ''">Please enter a value!</p>
```

## Attribute Directives

Cambian la apariencia o el comportamiento de un elemento del DOM de la página web.

### ngClass

La directiva aplicará la clase **'btn-primary'** si el objeto **newItem** no está vacío, en caso contrario aplicará **'btn-default'**. Es una directiva no estructural

```
<button
  (click)="onAddItem()"
  class="btn"
  [ngClass]="{'btn-primary': newItem !== '', 'btn-default' : newItem
=== ''}">>Add Item</button>
```

### ngStyle

La directiva nos permite cambiar el estilo de la propiedad **'background-color'** para los elementos pares e impares, cabe destacar que la palabra **"index"** en la directiva **ngFor** representa el índice del bucle **for**. Es una directiva no estructural.

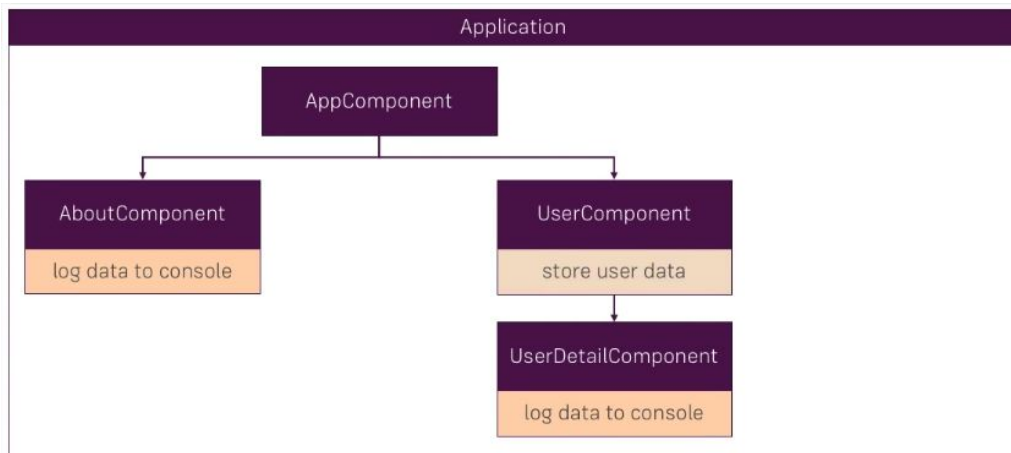
```
<ul class="list-group">
  <li
    class="list-group-item"
    *ngFor="let it of items; let i = index"
    [ngStyle]="{'background-color': i % 2 === 0 ? 'yellow' : 'blue'}">{{
it }}</li>
</ul>
```

[Ir al ejemplo "Uso de directivas estructurales y de atributos"](#)

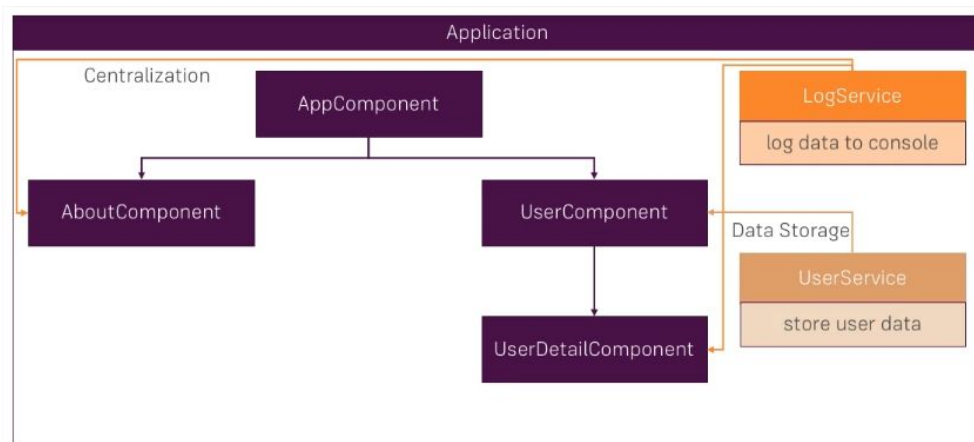
[Ir al ejemplo "Caso práctico 1"](#)

## Services

Sin el uso de “Services” se puede llegar fácilmente a repetir código en diferentes partes y a tener problemas con la información guardada que sea diferente dependiendo del componente.



Los “Services” nos permiten centralizar funcionalidades que se usan en diferentes partes del código y información que podemos necesitar en diferentes partes de la web.



Para usar un Service en cualquier componente de nuestro proyecto, tenemos que ponerlo como “providers” dentro del fichero **app.module.ts** (previamente se ha de hacer el “import”).

```
@NgModule({
  declarations: [
    AppComponent,
    TabsComponent,
    ListComponent,
    ItemComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [StarWarsService, LogService],
  bootstrap: [AppComponent]
})
```

Y para usarlo dentro de un componente se declara una propiedad del componente del tipo del servicio que queremos usar y lo pasamos como argumento por el constructor. (previamente se ha de hacer el “import”)

```
export class ItemComponent implements OnInit {  
  @Input() character;  
  swService: StarWarsService;  
  
  constructor(swService: StarWarsService) {  
    this.swService = swService;  
  }  
}
```

Una vez hecho, se puede usar en el componente:

```
onAssign(newSide) {  
  this.swService.onSideChosen({name: this.character.name, side: newSide});  
}
```

## Decorator @Injectable()

Permite a una clase que se le puedan inyectar dependencias, como es el caso de la clase “Service”, que en su defecto Angular no lo permite.

[Ir al ejemplo “Caso práctico 1 pasado a Services”](#)

## Forms (Template-driven)

### FormsModule

El **FormsModule** es el que nos permitirá utilizar todas las características de Angular para formularios. Para usarlo se tiene que importar en el **app.module.ts** y añadir en el array de “imports”.



```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { FormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6 import { TabsComponent } from './tabs/tabs.component';
7 import { ListComponent } from './list/list.component';
8 import { ItemComponent } from './item/item.component';
9 import { StarWarsService } from './star-wars.service';
10 import { LogService } from './log.service';
11 import { CreateCharacterComponent } from './create-character/cre
12
13 @NgModule({
14   declarations: [
15     AppComponent,
16     TabsComponent,
17     ListComponent,
18     ItemComponent,
19     CreateCharacterComponent,
20   ],
21   imports: [
22     BrowserModule,
23     FormsModule
24   ],
25   providers: [StarWarsService, LogService],
26   bootstrap: [AppComponent]
27 })
28 export class AppModule { }
29
```

### Template reference variable

Una “template reference variable” a menudo hace referencia a un elemento del DOM dentro de una plantilla (código html de un componente). También puede referenciar a una directiva (que contenga un componente), un elemento, TemplateRef, o un componente web. De momento veremos elementos de DOM y directivas.

### Ejemplo de elemento del DOM

```
<input #phone placeholder="phone number" />

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler
-->
<button (click)="callPhone(phone.value)">Call</button>
```

### Ejemplo de directiva

```
<form #itemForm="ngForm" (ngSubmit)="onSubmit(itemForm)">
  <label for="name"> Name
    <input class="form-control" name="name" ngModel required />
  </label>
  <button type="submit">Submit</button>
</form>

<div [hidden]="!itemForm.form.valid">
  <p>{{ submitMessage }}</p>
</div>
```

## ngForm Directive

Angular automáticamente crea y vincula una directiva **ngForm** a la etiqueta `<form>`. La directiva añade funcionalidades extra al formulario. Contiene los controles que se han creado para los elementos con la **directiva ngModel** y el **atributo name**, y monitoriza sus propiedades incluida su validación. Adicionalmente tiene su propiedad `valid` que solamente será verdadera si todos los otros controles que contiene lo son.

## ngModel Directive

La directiva `ngModel` no solamente se usa para el “Two-way binding” (valor y evento), sino que en los formularios se usa para registrar y saber el estado de los controles.

State	Class if true	Class if false
The control has been visited.	<code>ng-touched</code>	<code>ng-untouched</code>
The control's value has changed.	<code>ng-dirty</code>	<code>ng-pristine</code>
The control's value is valid.	<code>ng-valid</code>	<code>ng-invalid</code>

<https://angular.io/guide/forms#track-control-state-and-validity-with-ngmodel>



## Validators

Angular tiene los siguientes validadores:

Validador	Descripción	Ejemplo
required	El campo no puede estar vacío	<code>&lt;input type="text" required&gt;</code>
email	Comprueba si está escrito correctamente	<code>&lt;input type="email" email&gt;</code>
minlength	Checks for minimum length	<code>&lt;input type="text" minlength="5"&gt;</code>
maxlength	Checks for maximum length	<code>&lt;input type="text" maxlength="5"&gt;</code>
pattern	Checks for a certain RegEx pattern	<code>&lt;input type="text" pattern="[a-zA-Z ]*"&gt;</code>

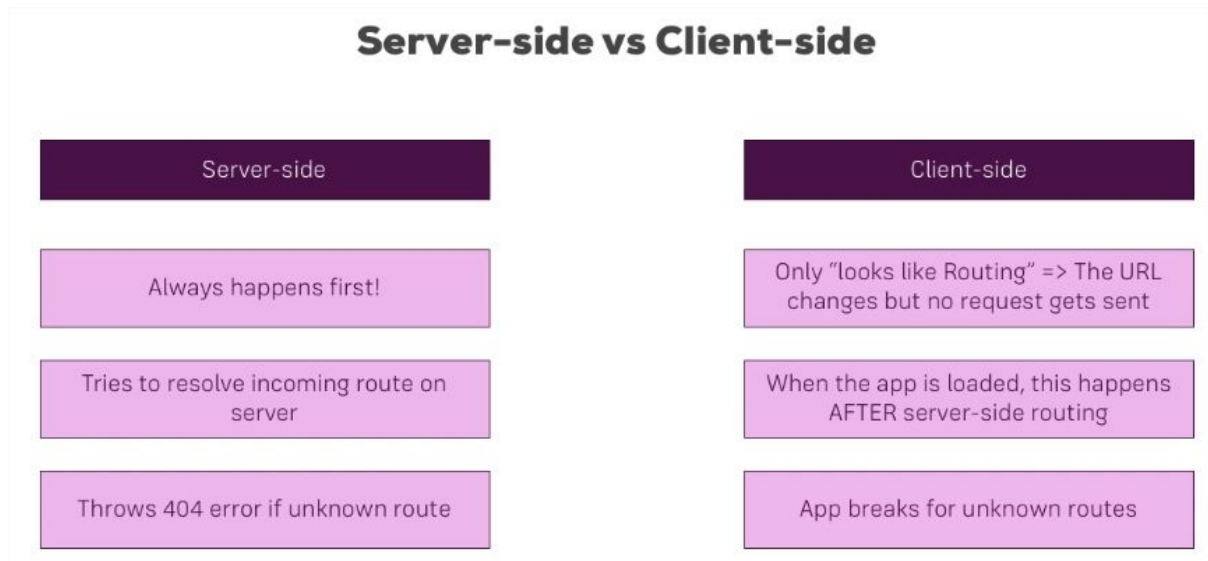
<https://angular.io/guide/form-validation>

[Ir al ejemplo "Caso práctico 1 con Formulario"](#)

[Ir al ejemplo "Formulario de registro"](#)

# Routing

## Server-side vs Client-side



## RouterModule

Para usar el **RouterModule** se tiene que importar en el **app.module.ts** y añadir en el array de "imports", al igual que el **FormsModule**. Además habrá que especificar las rutas de nuestra página web tal y como en la figura:

```
14
15 const routes = [
16   { path: '', component: TabsComponent },
17   { path: 'new-character', component: CreateCharacterComponent }
18 ];
19
20 @NgModule({
21   declarations: [
22     AppComponent,
23     TabsComponent,
24     ListComponent,
25     ItemComponent,
26     CreateCharacterComponent,
27     HeaderComponent
28   ],
29   imports: [
30     BrowserModule,
31     FormsModule,
32     RouterModule.forRoot(routes)
33   ],
34   providers: [StarWarsService, LogService],
35   bootstrap: [AppComponent]
36 })
```

La primera ruta definida es la que tendremos por defecto y el componente que en ella se mostrará, la siguiente ruta ya tiene definido un **path** que se añadirá a la dirección url de nuestra página web (P.ej. <http://localhost:4200/new-character>)

Una vez configuradas habrá que decirle a Angular donde queremos que se muestre el contenido de cada una de nuestras páginas en el **app.component.html**:

```
app.component.html x
1  <div class="container">
2    <div class="row">
3      <div class="col-xs-12 col-sm-6">
4        <app-header></app-header>
5      </div>
6    </div>
7    <hr>
8    <div class="row">
9      <div class="col-xs-12">
10       <router-outlet></router-outlet>
11     </div>
12   </div>
13 </div>
```

Una vez configuradas las rutas y especificado el lugar donde queremos que se muestre el contenido, tendremos que hacer que funcione al hacer click encima del enlace correspondiente:

```
<ul class="nav nav-pills">
  <li class="nav-item">
    <a class="nav-link"
      routerLink="/"
      routerLinkActive="active">Star Wars Characters</a>
  </li>
  <li class="nav-item">
    <a class="nav-link"
      routerLink="/new-character"
      routerLinkActive="active">New character</a>
  </li>
</ul>
```

La directiva **routerLink** nos permite añadir una cadena de texto para especificar la ruta que queremos que se muestre al hacer click. Mientras que la directiva **routerLinkActive** nos añadirá en este caso la clase de estilo **"active"** (puede ser cualquier otra clase), cuando el usuario apriete en el enlace.

## Redireccionamientos

Para evitar errores en que nuestra aplicación se rompa por culpa de rutas no mapeadas, existe una opción que **debe ir al final del array de rutas en app.module.ts**, que nos permitirá en caso de no haber encontrado la ruta, nos redirija a la página que tenemos por defecto. Es importante ponerlo al final, ya que en caso contrario las rutas por detrás de esta nunca se mostrarían.

```
const routes = [  
  { path: '', component: TabsComponent },  
  { path: 'new-character', component: CreateCharacterComponent },  
  { path: '**', redirectTo: '/' }  
];
```

## Child Routes & Parámetros

En algún momento nos interesará hacer subdirecciones de una misma página para filtrar por ejemplo resultados o simplemente para dividir el contenido en secciones. Para hacerlo, en el **app.module.ts** tendremos que hacer lo siguiente:

```
// Si hay children (subrutas de una ruta) el "component" es obligatorio que sea diferente de el del padre  
// Y además el componente padre tiene que tener un <router-outlet>  
  
const routes = [  
  { path: 'register', component: RegistroUsuarioComponent },  
  { path: 'users', component: SelectorUsuariosComponent, children: [  
    { path: '', redirectTo: 'all', pathMatch: 'full' },  
    { path: ':type', component: ListaUsuariosComponent }  
  ] },  
  { path: 'updateuser', component: ModificarUsuarioComponent },  
  { path: '**', redirectTo: '/register' } // Hacer Dashboard welcome screen  
];
```

Donde vemos “:type” es una variable que usaremos para mostrar la subcategorías, pondremos la dirección completa en el **routerLink**, en este caso en **SelectorUsuariosComponent**, como en la figura siguiente:

```
<li class="nav-item">  
  <a class="text-sm-center nav-link"  
    routerLink="/users/administradores"  
    routerLinkActive="active">Administradores</a>  
</li>
```

## ActivatedRoute

Mediante la inyección de dependencias de **ActivatedRoute**, crearemos un **Observable** que en cuanto los parámetros que reciba este componente cambien, le pedirá al **Service** la lista de usuarios que respondan al parámetro que tenemos. Así de esta forma veremos un resultado filtrado por el parámetro que nos han pasado.

```
import { Component, OnInit } from '@angular/core';
import { GestorUsuariosService } from '../gestor-usuarios.service';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-lista-usuarios',
  templateUrl: './lista-usuarios.component.html',
  styleUrls: ['./lista-usuarios.component.css']
})
export class ListaUsuariosComponent implements OnInit {

  users = [];
  userManager: GestorUsuariosService;
  activatedRoute: ActivatedRoute;

  constructor(userManager: GestorUsuariosService, activatedRoute: ActivatedRoute) {
    this.userManager = userManager;
    this.activatedRoute = activatedRoute;
  }

  ngOnInit() {
    this.activatedRoute.params.subscribe(
      (params) => {
        this.users = this.userManager.getUsers(params.type);
      }
    );
  }
}
```

Por último, si deseamos hacer un cambio de página mediante código en typescript sin depender del código html, podemos usar la inyección de dependencia de **Router** para hacerlo, y a la vez pasar parámetros que podamos leer en la página de destino.



## Router (Envío de Datos)

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

@Component({
  selector: 'app-item-lista-usuarios',
  templateUrl: './item-lista-usuarios.component.html',
  styleUrls: ['./item-lista-usuarios.component.css']
})
export class ItemListaUsuariosComponent implements OnInit {

  @Input() user;

  userManager: GestorUsuariosService;
  router: Router;

  constructor(userManager: GestorUsuariosService, router: Router) {
    this.userManager = userManager;
    this.router = router;
  }

  ngOnInit() {
  }

  deleteUser() {
    this.userManager.deleteUser(this.user);
  }

  updateUser() {
    this.router.navigate(['updateuser', this.user]);
  }
}
```

## Router (Lectura de Datos)

```
constructor(route: ActivatedRoute, router: Router) {
  this.route = route;
  this.router = router;
}

ngOnInit() {
  this.user = {id: this.route.snapshot.paramMap.get('id'),
    name: this.route.snapshot.paramMap.get('name'),
    surname: this.route.snapshot.paramMap.get('surname'),
    email: this.route.snapshot.paramMap.get('email'),
    role: this.route.snapshot.paramMap.get('role')};
}
```

[Ir al ejemplo "Routers caso práctico 1"](#)

[Ir al ejemplo "Routers caso práctico 2"](#)

## Observables “subscribe()”

Un **Observable** es un fragmento de código que espera hasta que se cumpla una condición para ejecutarse. Normalmente cuando se asignan sobre un parámetro, se suele ejecutar el código cuando dicho parámetro cambia.

Cuando se realiza un “**.subscribe**” sobre un parámetro, en este caso “**params**” se configura un disparador de tal forma que cuando el parámetro “**params**” cambie su valor, se ejecuten las funciones siguientes:

```
const observer = {
  next: params => { /* Your Code */ }, // When the attribute "params" changes
  error: err => { /* Your Code */ }, // When an error happens
  complete: () => { /* Your Code */ }, // When observable finishes
};

this.route.params.subscribe(observer);
```

Cada una se ejecuta tal y como se describe en la imagen. Ninguna de ellas tiene ningún tipo de retorno.

## RxJS

Para empezar a trabajar la reactividad de nuestra aplicación y que reaccione mejor de lo que actualmente lo hace, debemos instalar RxJS.

**npm install rxjs-compat --save**

## Subject (Observable)

Para crear nuestros propios observables, podemos usar de **RxJS** la clase **Subject**, tal y como vemos en el ejemplo, creamos una variable del tipo **Subject** que no requiere que se le pase ningún parámetro por argumento:

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

import { LogService } from './log.service';

@Injectable()
export class StarWarsService {
  private characters = [
    { name: 'Luke Skywalker', side: '' },
    { name: 'Darth Vader', side: '' }
  ];
  private logService: LogService;
  charactersChanged = new Subject<void>();
```

Para que se ejecute el **Subject** que hemos creado, tenemos que hacer referencia a la llamada de la imagen a continuación. En este caso queremos que al cambiar un personaje de “lado de la fuerza” se ejecute el código.

```
onSideChosen(charInfo) {  
  const pos = this.characters.findIndex((char) => {  
    return char.name === charInfo.name;  
  });  
  
  this.characters[pos].side = charInfo.side;  
  this.charactersChanged.next();  
  this.logService.writeLog('Changed side of ' + charI  
}
```

Igual que un **Observable** hay que realizar un “**.subscribe**” para determinar el bloque de código que hay que ejecutar, una vez se cumpla la condición que queramos. En la clase **list-component** si un personaje cambia de “lado de la fuerza” recargamos la lista de personajes de la lista en la que estamos actualmente.

```
ngOnInit() {  
  this.activatedRoute.params.subscribe(  
    (params) => {  
      this.characters = this.swService.getCharacters(params.side);  
      this.loadedSide = params.side;  
    }  
  );  
  
  this.swService.charactersChanged.subscribe(  
    () => {  
      this.characters = this.swService.getCharacters(this.loadedSide);  
    }  
  );  
}
```

## Gestión de memoria

Al utilizar la clase **Subject** de **RxJS** tenemos que gestionar manualmente la eliminación de todos los “**subscribe**” que hagamos como indica la siguiente imagen:

```
export class ListComponent implements OnInit, OnDestroy {  
  subscription;  
  
  ngOnInit() {  
    this.subscription = this.swService.charactersChanged.subscribe(  
      () => {  
        this.characters = this.swService.getCharacters(this.loadedSide);  
      }  
    );  
  }  
  
  ngOnDestroy() {  
    this.subscription.unsubscribe();  
  }  
}
```

[Ir al ejemplo “RxJS caso práctico 1”](#)



## Http Client

Primero hay que importar en el **app.module.ts** el siguiente módulo:

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

```
imports: [  
  BrowserModule,  
  FormsModule,  
  RouterModule.forRoot(routes),  
  HttpClientModule  
],
```

Luego inyectamos la dependencia del módulo donde necesitemos hacer las peticiones, como por ejemplo en nuestro servicio, para que obtenga los personajes del servidor:

```
private http: HttpClient;  
  
charactersChanged = new Subject<void>();  
  
constructor(logService: LogService, http: HttpClient) {  
  this.logService = logService;  
  this.http = http;  
}
```

## GET

Para realizar una petición GET al servidor tenemos que crear un objeto que haga de **Observer**, como el que tenemos en la imagen y pasarlo al subscribe del get.

```
const getObserver = {  
  next: value => {  
    console.log('GET: ');  
    console.log(value);  
  },  
  error: err => {  
    console.log('GET ERROR: ');  
    console.log(err);  
  },  
  complete: () => {  
    console.log('GET Finished !!');  
  },  
};  
  
this.http.get(this.getUrl).subscribe(getObserver);
```

## POST

El post se hace de la misma forma que el GET pero añadiendo la información en un objeto que se le pasa a la función:

```
const body = { id: 0, name: 'Perico' };
const postObserver = {
  next: value => {
    console.log('POST: ');
    console.log(value);
  },
  error: err => {
    console.log('POST ERROR: ');
    console.log(err);
  },
  complete: () => {
    console.log('POST Finished !!');
  },
};

this.http.post(this.postUrl, body).subscribe(postObserver);
```

Para hacer pruebas contra un servidor que acepte peticiones GET y POST podemos utilizar <http://httpbin.org/>, para las peticiones GET <http://httpbin.org/get> y para las POST <http://httpbin.org/post>.

Si nuestro servidor PHP está en local, deberemos añadir las siguientes líneas al inicio del fichero .php para que nos devuelva respuesta:

```
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: X-API-KEY, Origin, X-Requested-With,
Content-Type, Accept, Access-Control-Request-Method");
header("Access-Control-Allow-Methods: GET, POST, OPTIONS, PUT, DELETE");
header("Allow: GET, POST, OPTIONS, PUT, DELETE");
```

También deberemos devolver desde nuestro servidor, un JSON que sea válido, para ello podemos comprobar en la siguiente página si es correcto o no nuestro JSON:

<https://jsonformatter.curiousconcept.com/>

Adicionalmente para tratar desde el PHP el objeto enviado desde Angular necesitaremos la siguiente línea de código que nos convierte el objeto en JSON en PHP:

```
$json = file_get_contents('php://input');
```

Y en caso de querer tratar ese objeto en PHP: *\$obj = json\_decode(\$json, TRUE);*

## Custom Attribute Directives

Cuando creamos una **Attribute Directive** es para poder cambiar o modificar cualquiera de los atributos de una etiqueta. Para ello al crear una nueva directiva de atributo tenemos que acceder al elemento desde la que se ha llamado con **ElementRef**.

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

@Directive({
  selector: '[appInvisibleInk]'
})
export class InvisibleInkDirective {

  // @Input ('alias') -> the 'alias' can be used instead of "text" from outside
  @Input('appInvisibleInk') textColor: string;
  @Input() defaultTextColor: string;

  constructor(private elem: ElementRef) {
    elem.nativeElement.style.color = 'white';
    elem.nativeElement.style.border = '2px solid black';
  }
}
```

Tal y como vemos en la imagen, importamos **ElementRef** y lo inyectamos en el constructor. Una vez hecho podemos manipular los estilos del elemento a nuestro gusto. Al igual que en los componentes también podemos pasar parámetros desde fuera de la directiva. En este caso podemos usar un “alias” dentro del **Input('alias') Decorator** para que desde fuera podamos acceder con el nombre del alias en lugar de con el de la variable.

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

@Directive({
  selector: '[appInvisibleInk]'
})
export class InvisibleInkDirective {

  // @Input ('alias') -> the 'alias' can be used instead of "text" from outside
  @Input('appInvisibleInk') textColor: string;
  @Input() defaultTextColor: string;
}
```

Por último, podemos usar el **Decorator @HostListener** para añadir comportamientos específicos cuando cierto tipo de eventos ocurran.

```

@HostListener('mouseenter')
onMouseEnter() {
  this.showText(true);
}

@HostListener('mouseleave')
onMouseLeave() {
  this.showText(false);
}

showText(show: boolean) {
  const color = ((show) ? this.textColor || this.defaultTextColor || 'black' : 'white');
  this.elem.nativeElement.style.color = color;
}

```

## Structural Directives NgFor (Avanzado)

```

<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackFunction" [class.odd]="odd">
  Indice: {{i}} ## Elemento: {{hero.name}} ## Es impar ? {{odd}}
</div>

```

En este caso podemos ver más parámetros que describimos a continuación:

- “**let i=index**”: La variable **i** contiene el índice del elemento dentro del array.
- “**let odd=odd**”: La variable **odd** contiene un booleano si el elemento es par o no.
- “**trackBy: trackFunction**”: Por defecto, Angular usa una instancia del propio elemento a encontrar dentro del array para detectar si han habido cambios. Esto puede ser altamente costoso en rendimiento para arrays muy grandes. Por eso nos permite pasarle una función, “**trackFunction**” para este ejemplo, que recibe 2 parámetros: el índice del elemento y la instancia del objeto. En estos casos suele ser mejor devolver el índice para que pueda acceder más fácilmente o depende del caso el identificador del objeto.

```

trackFunction(index, object) {
  if (!object) {
    return null;
  }

  return index;
}

```

## Structural Directives NgSwitch

La directiva `ngSwitch` nos permite cambiar un componente por otro de los que tengamos en la lista en función del valor de la variable que evaluemos en el switch, en este caso la variable es la propiedad “**emotion**” del objeto “**hero**”:

```
<div [ngSwitch]="hero?.emotion">
  <app-happy-hero      *ngSwitchCase="'happy'"      [hero]="hero"></app-happy-hero>
  <app-sad-hero        *ngSwitchCase="'sad'"          [hero]="hero"></app-sad-hero>
  <app-confused-hero   *ngSwitchCase="'confused'"     [hero]="hero"></app-confused-hero>
  <app-unknown-hero    *ngSwitchDefault               [hero]="hero"></app-unknown-hero>
</div>
<button (click)="switchHeroState()">Switch Emotion</button>
<p>Current emotion: {{ hero.emotion }}</p>
```

[Ir al ejemplo "Advanced directives and custom attribute directives"](#)

## Angular Animations

Para empezar a hacer animaciones en nuestra página, primero debemos importar el módulo de animaciones de Angular:

```
app.module.ts x
src > app > app.module.ts > ...
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
4
5 import { AppComponent } from './app.component';
6
7 @NgModule({
8   declarations: [
9     AppComponent
10  ],
11   imports: [
12     BrowserModule,
13     BrowserAnimationsModule
14  ],
```

Una vez hecho esto, tenemos que importar en el componente las funciones que vamos a usar del módulo de animaciones. Una vez hecho, tenemos que definir los estados de lo que vamos a animar ('open' y 'close'), definir las transiciones entre ellos ('open-closed' y 'closed-open') y por último el nombre del trigger ('openCloseTrigger').

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

@Component({
  selector: 'app-open-close',
  templateUrl: './open-close.component.html',
  styleUrls: ['./open-close.component.css'],
  animations: [
    trigger('openCloseTrigger', [
      // ...
      state('open', style({
        height: '200px',
        opacity: 1,
        backgroundColor: 'yellow'
      })),
      state('closed', style({
        height: '100px',
        opacity: 0.5,
        backgroundColor: 'green'
      })),
      transition('open => closed', [
        animate('1s')
      ]),
      transition('closed => open', [
        animate('0.5s')
      ])
    ]
  ],
})
```



Por supuesto, esto es un ejemplo de los múltiples que se pueden hacer y no todos tienen que seguir este patrón. Por último añadimos el atributo con el nombre del trigger y le asignamos la condición por la que se disparará una transición u otra.

```
<div [@openCloseTrigger]="isOpen ? 'open' : 'closed'" class="open-close-container" (click)="toggle()">
  <p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>
</div>
```

## Lista de funciones

Function name	What it does
<code>trigger()</code>	Kicks off the animation and serves as a container for all other animation function calls. HTML template binds to <code>triggerName</code> . Use the first argument to declare a unique trigger name. Uses array syntax.
<code>style()</code>	Defines one or more CSS styles to use in animations. Controls the visual appearance of HTML elements during animations. Uses object syntax.
<code>state()</code>	Creates a named set of CSS styles that should be applied on successful transition to a given state. The state can then be referenced by name within other animation functions.
<code>animate()</code>	Specifies the timing information for a transition. Optional values for <code>delay</code> and <code>easing</code> . Can contain <code>style()</code> calls within.
<code>transition()</code>	Defines the animation sequence between two named states. Uses array syntax.
<code>keyframes()</code>	Allows a sequential change between styles within a specified time interval. Use within <code>animate()</code> . Can include multiple <code>style()</code> calls within each <code>keyframe()</code> . Uses array syntax.
<code>group()</code>	Specifies a group of animation steps ( <i>inner animations</i> ) to be run in parallel. Animation continues only after all inner animation steps have completed. Used within <code>sequence()</code> or <code>transition()</code> .

<code>query()</code>	Use to find one or more inner HTML elements within the current element.
<code>sequence()</code>	Specifies a list of animation steps that are run sequentially, one by one.
<code>stagger()</code>	Staggeres the starting time for animations for multiple elements.
<code>animation()</code>	Produces a reusable animation that can be invoked from elsewhere. Used together with <code>useAnimation()</code> .
<code>useAnimation()</code>	Activates a reusable animation. Used with <code>animation()</code> .
<code>animateChild()</code>	Allows animations on child components to be run within the same timeframe as the parent.

## Recursos para el diseño y creación de animaciones

<https://material.io/design/motion/speed.html#easing>

[Ir al ejemplo "Simple Animation"](#)

## Deployment

Para preparar el proyecto y poderlo subir a un servidor, tendremos que ejecutar desde el directorio de nuestro proyecto:

```
ng build --prod
```

Una vez finalice, tendremos nuestro proyecto generado en la carpeta **dist** dentro de la actual de nuestro proyecto. Dentro de esa carpeta encontraremos los archivos que tenemos que subir a nuestro servidor.



## Recursos

En la siguiente página podemos encontrar componentes ya creados por desarrolladores de Angular que nos pueden ayudar a realizar nuestra página web:

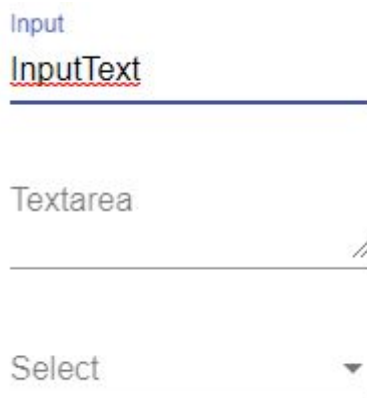
<https://material.angular.io/>

Para poder usar los ejemplos, hace falta instalar las dependencias con la siguiente línea de código:

```
npm install --save @angular/material @angular/cdk @angular/animations
```

Algunos componentes necesitarán el **BrowserAnimationModule**, para poder usarlo hace falta seguir el proceso ya descrito en la sección [Angular Animations](#). Por último, solamente hará falta importar en **app.module.ts** el módulo del componente que queramos usar, por ejemplo la siguiente imagen de la izquierda se muestra un **Input**, **Textarea** y un **Select**. Para ello necesitaríamos los imports siguientes:

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatSelectModule } from '@angular/material/select';
import { MatInputModule } from '@angular/material/input';
```



```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MatFormFieldModule,
    MatSelectModule,
    MatInputModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

# Ejemplos

## Comunicación entre 2 componentes

player-input y player mediante el app.component.

- Directiva estructural ngIf
- Two-way Binding (Directiva ngModel)
- Event Binding
- Custom Events with: EventEmitter
- Decorators Input & Output

<https://drive.google.com/open?id=1dJ73kVJN7oB62DhHS9c9cd5uNV9Ql0gn>

## Uso de directivas estructurales y de atributos

- ngFor
- ngIf
- ngClass
- ngStyle

[https://drive.google.com/open?id=1mkJUfKJo9mUiku\\_S4prILLd61WLK3-Zy](https://drive.google.com/open?id=1mkJUfKJo9mUiku_S4prILLd61WLK3-Zy)

## Caso práctico 1

- Directivas
- Decorators Input & Output
- Two-way Binding (Directiva ngModel)
- Event Binding
- Custom Events with: EventEmitter

<https://drive.google.com/open?id=1GbuaL-HCpBPXiNc2UdETLqUZlmvUbJsH>

## Caso práctico 1 pasado a Services

<https://drive.google.com/open?id=1AQNs0WmJ-NjbTmdA14Alk1Y9qCi-q7H9ç>

## Caso práctico 1 con Formulario

<https://drive.google.com/open?id=19Cu2vhP9sFx3enpqlspKNOu0SamDyvhC>

## Formulario de registro

<https://drive.google.com/open?id=1rJGJoFcJwCkbyWOKhEJixNEztWV9Nx->

## Routers caso práctico 1

<https://drive.google.com/open?id=1QsPhijls4ECrgF8BanoiFrY1iq7ciWBR>

## Routers caso práctico 2

[https://drive.google.com/open?id=1tpaO7q1yEeSe5R\\_fZdcIT82iLOYUnuPo](https://drive.google.com/open?id=1tpaO7q1yEeSe5R_fZdcIT82iLOYUnuPo)

## RxJS caso práctico 1

<https://drive.google.com/open?id=1bRPgu25OJHYmDLth5p5mGpHa8MhJ4M02>

## Angular Role Game Project

Proyecto que recoge casi todos los puntos anteriormente vistos.

<https://drive.google.com/open?id=1PqDZZiDIFhVFv0gWP-0OJg3njW48pJD>

## Advanced directives and custom attribute directives

[https://drive.google.com/open?id=1AB9AabmKVxezscFplJFf\\_nuj56Fi8JvE](https://drive.google.com/open?id=1AB9AabmKVxezscFplJFf_nuj56Fi8JvE)

## Simple Animation

<https://drive.google.com/open?id=1UM4YuSXjfMr1qulZYky3Y30xVb2mWzec>