BIENVENIDO

Javier Miguel
@javiermiguelg

KEEP CODING

Get started

Angular - Requerimientos

nodejs.org

npmjs.com

Angular - Herramientas

code.visualstudio.com

jetbrains.com/webstorm

Angular - Referencias

angular.io

typescriptlang.org

Angular - Get started

dudas & preguntas

Introducción

Angular - Introducción

¿Qué es Angular?

- Framework para crear aplicaciones con HTML y Javascript.
- Cross platform: Web, Native Mobile, Native Desktop.
- Ofrece herramientas nativas para el testeo del código.
- Tiene una comunidad realmente grande y activa.

Angular - Introducción

¿Cómo empezamos a trabajar con Angular?

Angular - Introducción

Boilerplate manual	`https://angular.io/docs/ts/latest/quickstart.html`
Boilerplate automático	`https://cli.angular.io`

Angular - Introducción

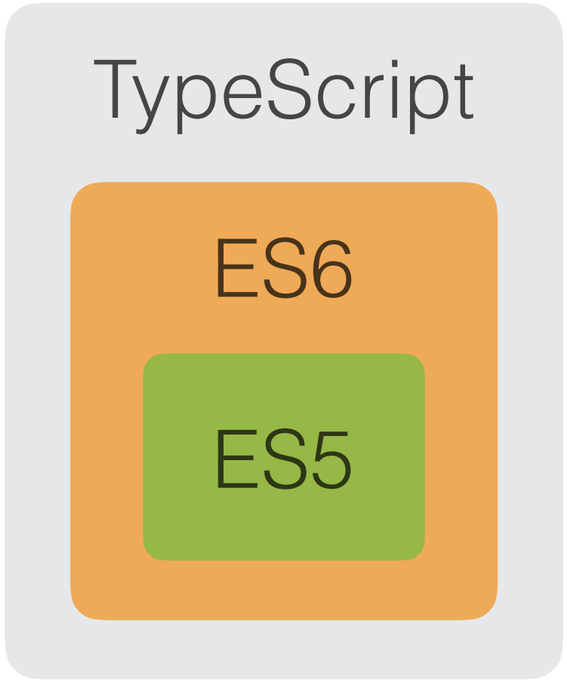
dudas & preguntas

TypeScript

Angular - TypeScript

- Es una extensión de Javascript.
- Añade características de POO: tipado de datos, clases, etc.
- Compila a Javascript (*transpile*).
- Corre en cualquier motor que soporte ECMAScript 3.

Angular - TypeScript



Angular - TypeScript

Tipado

```
// Type annotation.
let message: string = "Hello World!";

// Type inference.
let message = "Hello World!";

// Error.
message = 10;
```

Angular - TypeScript

Tipado

```
// Type annotation.
let message: string = "Hello World!";

// Type inference.
let message = "Hello World!";

// Error.
message = 10;
```

Podemos anotar el tipo de una variable usando : tras su definición.

Angular - TypeScript

Tipado

```
// Type annotation.
let message: string = "Hello World!";

// Type inference.
let message = "Hello World!"

// Error.
message = 10;
```

Si no anotamos el tipo, se infiere.

Angular - TypeScript

Tipado

```
// Type annotation.
let message: string = "Hello World!";

// Type inference.
let message = "Hello World!";

// Error.
message = 10
```

El tipo de una variable no puede cambiarse.

Angular - TypeScript

Type alias

```
type loggable = string | number | boolean;

let log = (data: loggable) => {
 console.log(`${new Date()} | ${data}`);
};
```

Angular - TypeScript

Type alias

```
type loggable = string | number | boolean;

let log = (data: loggable) => {
 console.log(`${new Date()} | ${data}`);
};
```

Usamos `type` para crear un nuevo alias.

Angular - TypeScript

Type alias

```
type loggable = string | number | boolean

let log = (data: loggable) => {
 console.log(`${new Date()} | ${data}`);
};
```

Podemos hacer uso de union types para definir el alias.

Angular - TypeScript

Interfaces

```
interface SuperHero {
 powers: string[];
 attack();
}

class Batman implements SuperHero { ... }
class Superman implements SuperHero { ... }
```

Angular - TypeScript

Interfaces

Creamos una interfaz con la palabra reservada `interface`.

```
interface SuperHero {
  powers: string[];
  attack();
}

class Batman implements SuperHero { ... }
class Superman implements SuperHero { ... }
```

Interfaces

Usamos `implements` para hacer que una clase implemente una interfaz.

```
interface SuperHero {
 powers: string[];
 attack();
}

class Batman implements SuperHero { ... }
class Superman implements SuperHero { ... }
```

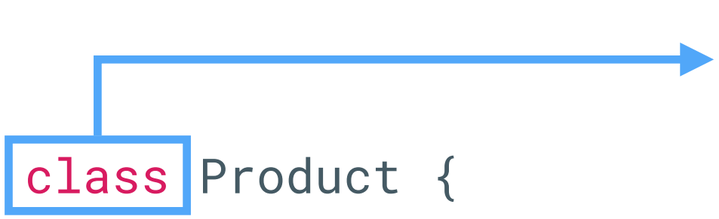
Angular - TypeScript

Clases

```
class Product {

 constructor(
   private id: number,
   public name: string) { }
}
```

Angular - TypeScript

Clases



Creamos una clase con la palabra reservada `class`.

```
class Product {

  constructor(
    private id: number,
    public name: string) { }
}
```

Angular - TypeScript

Clases

```
class Product {

  constructor(
    private id: number,
    public name: string) { }
}
```

Podemos crear automáticamente los atributos de una clase si establecemos el modificador de acceso en los parámetros del constructor.

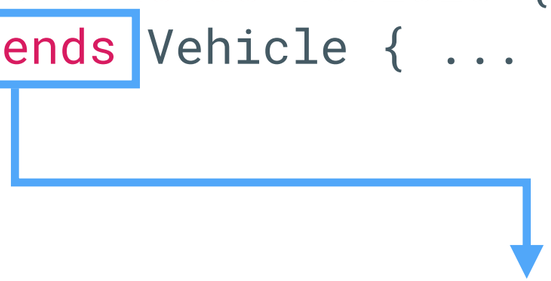
Angular - TypeScript

Herencia

```
class Vehicle { ... }

class Car extends Vehicle { ... }
class Motorcycle extends Vehicle { ... }
class Truck extends Vehicle { ... }
```

Herencia

```
class Vehicle { ... }

class Car extends Vehicle { ... }
class Motorcycle extends Vehicle { ... }
class Truck extends Vehicle { ... }
```



Con `extends` indicamos herencia de clases.

Angular - TypeScript

Genéricos

```
class List<T> {
 constructor(private items: T[] = []) { }
 ...
}

let products = new List<Product>();
let users = new List<User>();
```

Angular - TypeScript

Genéricos

```
class List<T> {
 constructor(private items: T[] = []) { }
 ...
}

let products = new List<Product>();
let users = new List<User>();
```

Usamos <T> para definir genéricos.

Angular - TypeScript

Genéricos

```
class List<T> {
 constructor(private items: T[] = []) { }
 ...
}

let products = new List<Product>();
let users = new List<User>();
```

Para usar los genéricos, indicamos el tipo entre <>.

Angular - TypeScript

Template Strings

```
let message = `${user.name} is ${user.age}`;
```

Angular - TypeScript

Template Strings

```
let message = `${user.name} is ${user.age}`;
```

Las cadenas de texto definidas con acento grave soportan interpolación de expresiones.

Angular - TypeScript

Template Strings

```
let message = `${user.name} is ${user.age}`;
```

Usaremos ${} para establecer expresiones a evaluar previamente a su interpolación en la cadena de texto.

Angular - TypeScript

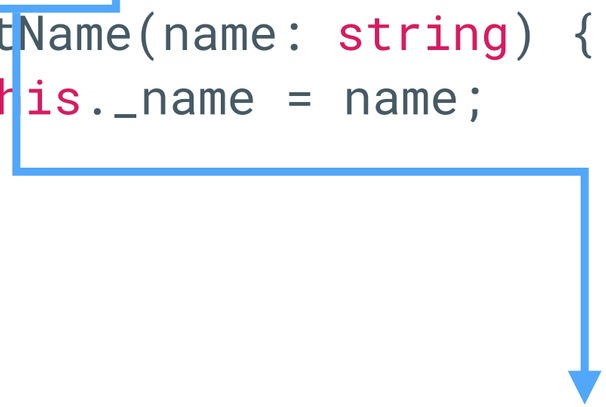
Decoradores

```
class User {

  @log()
  setName(name: string) {
    this._name = name;
  }
}
```

Angular - TypeScript

Decoradores

```
class User {

 @log()
  setName(name: string) {
    this._name = name;
  }
}
```



Con los decoradores podemos modificar el comportamiento de una clase, función o atributo en tiempo de ejecución.

Angular - TypeScript

Decoradores

```
class User {

 @log()
  setName(name: string) {
    this._name = name;
  }
}
```

Los decoradores son expresiones que resuelven una función; dicha función recibe información del contexto decorado.

Angular - TypeScript

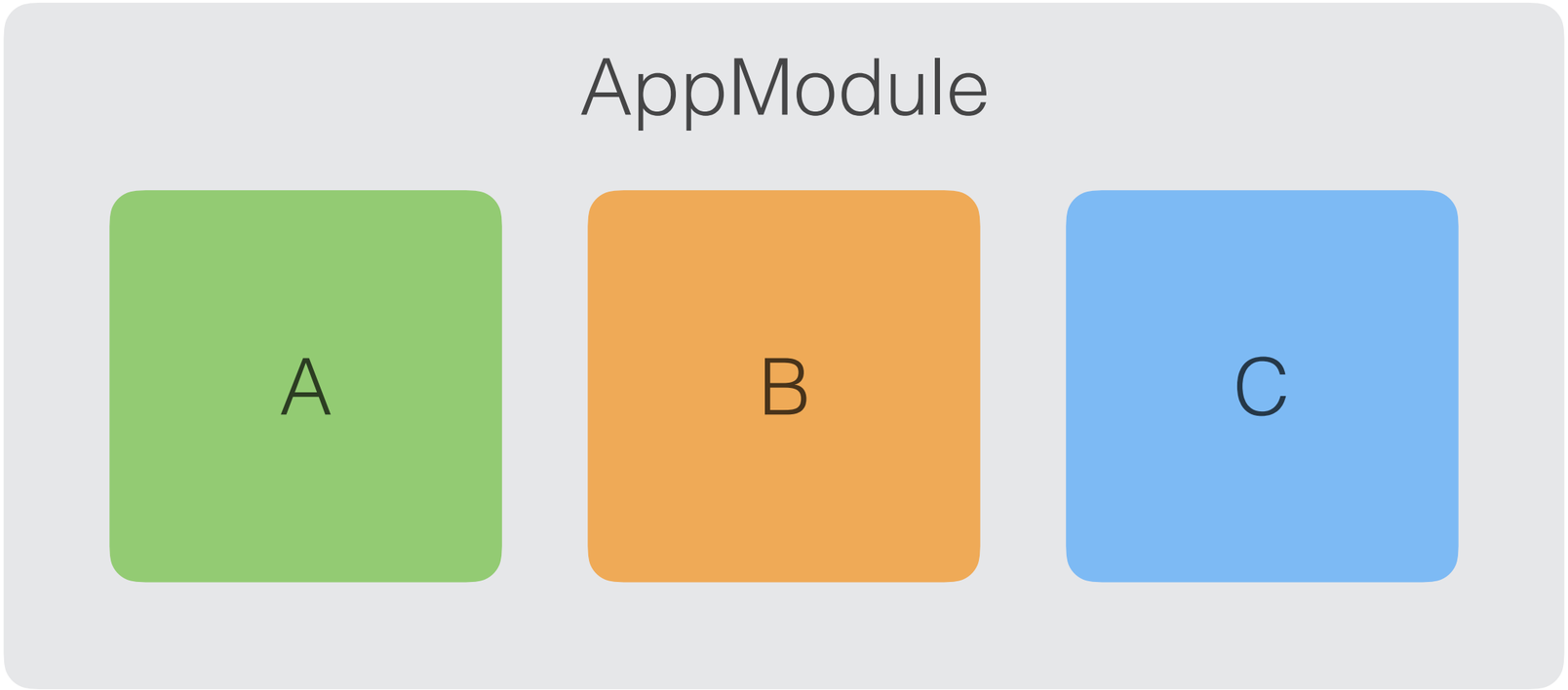
dudas & preguntas

Modules

Angular - Modules

- Sirven para organizar la app en bloques funcionales.
- Son clases decoradas con `@NgModule`.
- Toda app tiene al menos un módulo: `AppModule`.

Angular - Modules



Angular - Modules

Convención

Documento	`movies.module.ts`
Clase	`MoviesModule`

Importación

```
import { NgModule } from "@angular/core";
```

Angular - Modules

¿Qué metadatos indicamos en el decorador `@NgModule`?

Angular - Modules

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

Angular - Modules

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

En `imports` establecemos otros módulos de los cuales nuestra app depende.

Angular - Modules

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

En `declarations` establecemos todos los componentes, directivas y pipes que pertenecen a nuestra app.

Angular - Modules

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

En `providers` establecemos todos los servicios que pertenecen a nuestra app.

Angular - Modules

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

En `bootstrap` establecemos el componente raíz sobre el cuál se construye toda nuestra app.

Angular - Modules

dudas & preguntas

Binding

Angular - Binding

Interpolación

Angular - Binding

```
@Component({
  template: "<h1>{{ title }}</h1>"
})
export class UsersComponent {

  title: string = "Users";
}
```

Angular - Binding

```
@Component({
  template: "<h1>{{ title }}</h1>"
})
export class UsersComponent {

  title: string = "Users";
}
```

Con {{ x }} enlazamos un valor del componente en la vista correspondiente.

Angular - Binding

Enlace de propiedades

Angular - Binding

```
@Component({
  template: `<img [src]="user.image" />`
})
export class UserComponent {

  user: User = {
    id: 1,
    name: "John Doe",
    image: "images/john-doe.png"
  };
}
```

Angular - Binding

```
@Component({
  template: `<img [src]="user.image" />`
})
export class UserComponent {

  user: User = {
    id: 1,
    name: "John Doe",
    image: "images/john-doe.png"
  };
}
```

Con `[prop]` establecemos una expresión que debe resolverse previamente a su asignación a `prop`.

Angular - Binding

Enlace de clases

Angular - Binding

```
@Component({
  template: `<div [class.fav]="product.isFav"> ... </div>`
})
export class ProductComponent {

  product: Product = { isFav: true };
}
```

Angular - Binding

```
@Component({
  template: `<div [class.fav]="product.isFav" .. </div>`
})
export class ProductComponent {

  product: Product = { isFav: true };
}
```

Con `[class]` indicamos una clase CSS y asignamos una expresión como valor; solo si resuelve como `true`, la clase se añade al elemento.

Angular - Binding

Enlace de estilos

Angular - Binding

```
@Component({
  template: `<div [style.color]="getBoxColor(product)"> ... </div>`
})
export class ProductsComponent {

  getBoxColor(product: Product): string {
    return product.available ? "green" : "yellow";
  };
}
```

Angular - Binding

```
@Component({
  template: `<div [style.color]="getBoxColor(product)"> ... </div>`
})
export class ProductsComponent {

  getBoxColor(product: Product): string {
    return product.available ? "green" : "yellow";
  };
}
```

Con `[style]` indicamos un atributo CSS y asignamos una expresión; su resolución será el valor del atributo.

Angular - Binding

Enlace de eventos

Angular - Binding

```
@Component({
  template: `<button (click)="save()">Save</button>`
})
export class ProductComponent {

  save() {
    this._productService.save(this._product);
  };
}
```

Angular - Binding

```
@Component({
  template: `<button (click)="save()">Save</button>`
})
export class ProductComponent {

  save() {
    this._productService.save(this._product);
  };
}
```

Con `(event)` establecemos un manejador para dicho evento.

Angular - Binding

Enlace bidireccional

Angular - Binding

1. Importar `FormsModule` en nuestro módulo

Angular - Binding

```
import { FormsModule } from "@angular/forms";

@NgModule({
  ...
  import: [FormsModule]
  ...
})
export class AppModule { }
```

Angular - Binding

```
import { FormsModule } from "@angular/forms";

@NgModule({
  ...
  import: [FormsModule]
  ...
})
export class AppModule { }
```

El módulo `FormsModule` está en la librería `@angular/forms`.

Angular - Binding

```
import { FormsModule } from "@angular/forms";

@NgModule({
  ...
  import: [FormsModule]
  ...
})
export class AppModule { }
```

Lo añadimos al metadato `import` de nuestro módulo.

Angular - Binding

2. Ready to rock & roll !

Angular - Binding

```
@Component({
  template: `<input [(ngModel)]="product.name" />`
})
export class ProductComponent {

  product: Product;
}
```

Angular - Binding

```
@Component({
  template: `<input [(ngModel)]="product.name" />`
})
export class ProductComponent {

  product: Product;
}
```

Con `[(ngModel)]` establecemos un enlace bidireccional: el modelo se sincroniza desde la vista y la vista desde el modelo.

Angular - Binding

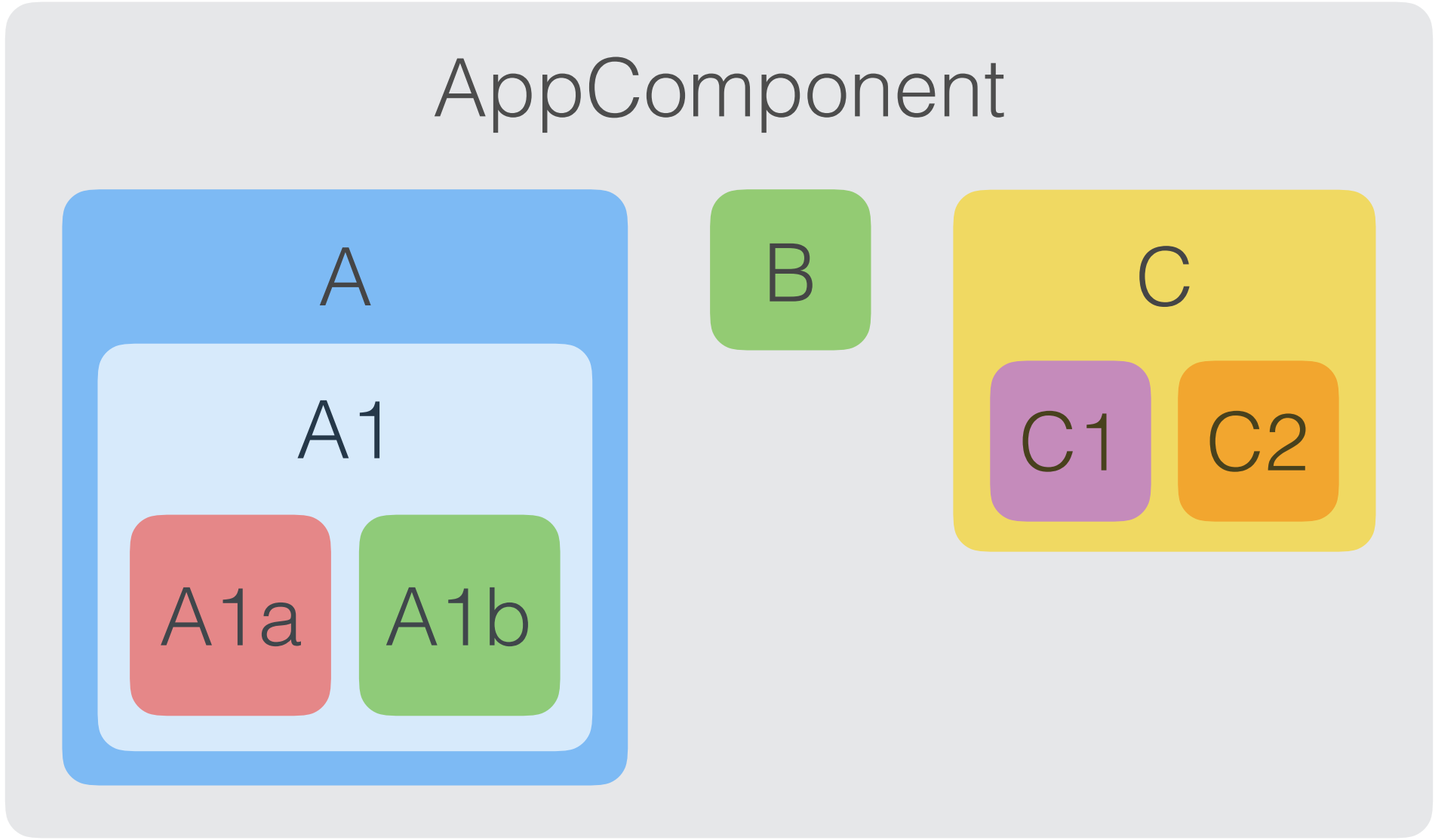
dudas & preguntas

Components

Angular - Components

- Son bloques de construcción de UI.
- Las apps son realmente árboles de componentes.
- Internamente son directivas con template.

Angular - Components



Angular - Components

Convención

Documento	`my-nav.component.ts`
Clase	`MyNavComponent`

Importación

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

Angular - Components

¿Qué metadatos indicamos en el decorador `@Component`?

Angular - Components

```
@Component({
  selector: "main",
  template: `
    <h1>Hello world!</h1>
    <h2>My first Angular 2 app</h2>
  `,
  styles: [
    "h1 { color: blue; }",
    "h2 { color: pink; }"
  ]
})
export class MainComponent { }
```

Angular - Components

```
@Component({
 selector: "main"
 template: `
  <h1>Hello world!</h1>
  <h2>My first Angular 2 app</h2>
 `,
 styles: [
  "h1 { color: blue; }",
  "h2 { color: pink; }"
 ]
})
export class MainComponent { }
```

En `selector` establecemos el nombre del elemento HTML en el cual Angular instanciará el componente.

Angular - Components

```
@Component({
  selector: "main",
  template: `
    <h1>Hello world!</h1>
    <h2>My first Angular 2 app</h2>
  `,
  styles: [
    "h1 { color: blue; }",
    "h2 { color: pink; }"
  ]
})
export class MainComponent { }
```

En `template` establecemos el HTML correspondiente a la vista del componente. Podemos usar `templateUrl` en su lugar para cargar un documento HTML externo.

Angular - Components

```
@Component({
  selector: "main",
  template: `
    <h1>Hello world!</h1>
    <h2>My first Angular 2 app</h2>
  `,
  styles: [
    "h1 { color: blue; }",
    "h2 { color: pink; }"
  ]
})
export class MainComponent { }
```

En `styles` establecemos los estilos CSS que aplican sobre el componente. Podemos usar `styleUrls` en su lugar para cargar uno o varios documentos CSS externos.

Angular - Components

Los componentes implementan algunos hooks en su ciclo de vida.

Angular - Components

Hook OnInit

Interfaz	`OnInit`
Método	`ngOnInit`
Propósito	Inicializar los datos del componente

Importación

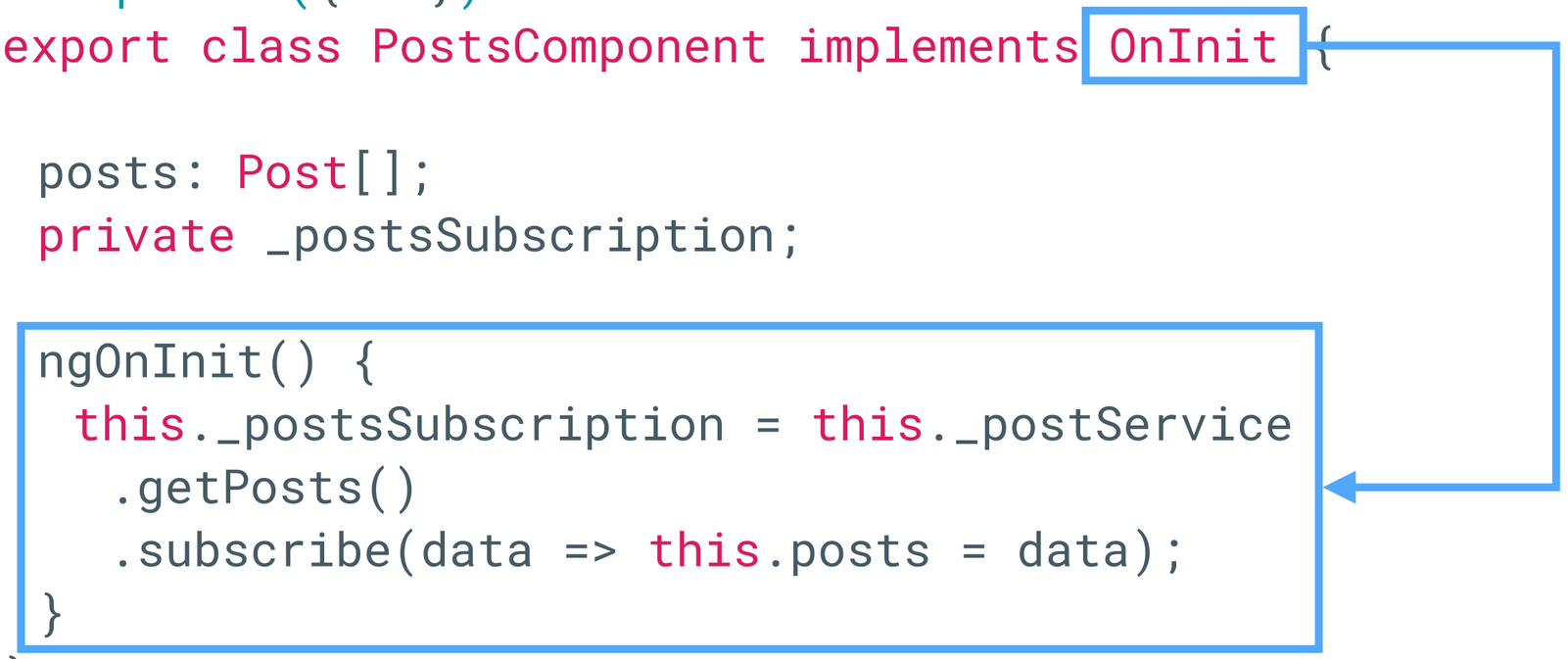
```
import { OnInit } from "@angular/core";
```

Angular - Components

```
@Component({...})
export class PostsComponent implements OnInit {

  posts: Post[];
  private _postsSubscription;

  ngOnInit() {
    this._postsSubscription = this._postService
      .getPosts()
      .subscribe(data => this.posts = data);
  }
}
```

Angular - Components

```
@Component({...})
export class PostsComponent implements OnInit {

  posts: Post[];
  private _postsSubscription;

  ngOnInit() {
    this._postsSubscription = this._postService
      .getPosts()
      .subscribe(data => this.posts = data);
  }
}
```



Angular - Components

Hook OnDestroy

Interfaz	`OnDestroy`
Método	`ngOnDestroy`
Propósito	Cancelación de eventos y suscripciones

Importación

```
import { OnDestroy } from "@angular/core";
```

Angular - Components

```
@Component({...})
export class PostsComponent implements OnDestroy {

  posts: Post[];
  private _postsSubscription;

  ngOnDestroy() {
    this._postsSubscription.unsubscribe();
  }
}
```

Angular - Components

```
@Component({...})
export class PostsComponent implements OnDestroy {

  posts: Post[];
  private _postsSubscription;

  ngOnDestroy() {
    this._postsSubscription.unsubscribe();
  }
}
```

Angular - Components

Los componentes pueden comunicarse entre ellos.

Angular - Components

Importación

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

Angular - Components

```
@Component({ selector: "posts" })
export class PostsComponent {

 @Input() postsPerPage: number;
}

@Component({ template: `<posts postsPerPage="20"></posts>` })
export class AppComponent { ... }
```

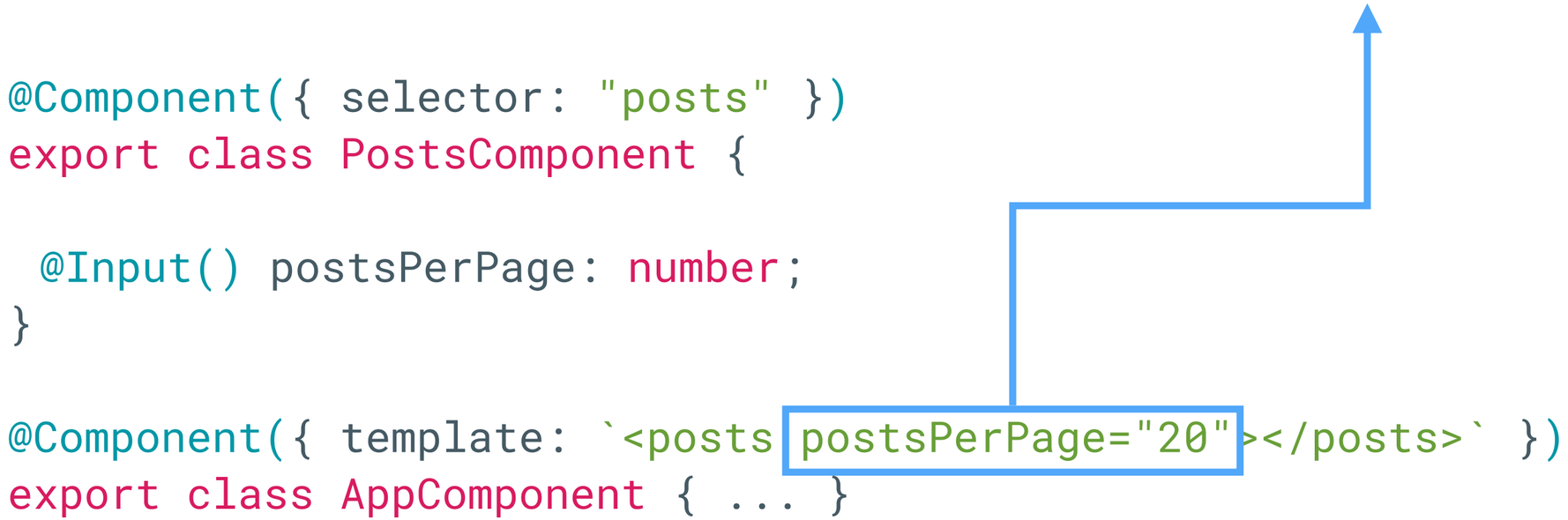
Angular - Components

Con el decorador `@Input` enlazamos datos de entrada en el componente.

```
@Component({ selector: "posts" })
export class PostsComponent {

  @Input() postsPerPage: number;
}

@Component({ template: `<posts postsPerPage="20"></posts>` })
export class AppComponent { ... }
```

Angular - Components

Para pasar un valor al componente, basta con hacer uso del atributo decorado con `@Input`.



```
@Component({ selector: "posts" })
export class PostsComponent {

 @Input() postsPerPage: number;
}

@Component({ template: `<posts postsPerPage="20"></posts>` })
export class AppComponent { ... }
```

Angular - Components

```
@Component({ selector: "posts" })
export class PostsComponent {

 private _page: number = 0;
 @Output() pageChanged: EventEmitter<number> = new EventEmitter();

 onNextPageButtonClick() { this.pageChanged.emit(++this._page); };
}

@Component({ template: `<posts (pageChanged)="onPage($event)"></posts>` })
export class AppComponent { ... }
```

Angular - Components

Con el decorador `@Output` enlazamos datos de salida en el componente.

```
@Component({ selector: "posts" })
export class PostsComponent {

 private _page: number = 0;
 @Output() pageChanged = EventEmitter<number> = new EventEmitter();

 onNextPageButtonClick() { this.pageChanged.emit(++this._page); };
}

@Component({ template: `<posts (pageChanged)="onPage($event)"></posts>` })
export class AppComponent { ... }
```

Angular - Components

Estableciendo el tipo `EventEmitter` indicamos que nuestro enlace tiene comportamiento de evento.

```
@Component({ selector: "posts" })
export class PostsComponent {

 private _page: number = 0;
 @Output() pageChanged: EventEmitter<number> = new EventEmitter();

 onNextPageButtonClick() { this.pageChanged.emit(++this._page); };
}

@Component({ template: `<posts (pageChanged)="onPage($event)"></posts>` })
export class AppComponent { ... }
```

Angular - Components

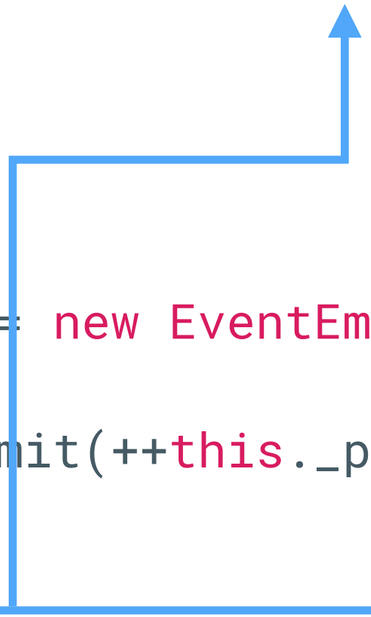
Con la función `emit()` notificamos datos a los subscriptores.

```
@Component({ selector: "posts" })
export class PostsComponent {

 private _page: number = 0;
 @Output() pageChanged: EventEmitter<number> = new EventEmitter();

 onNextPageButtonClick() { this.pageChanged.emit(++this._page); };
}

@Component({ template: `<posts (pageChanged)="onPage($event)"></posts>` })
export class AppComponent { ... }
```

Angular - Components

Para subscribirnos al evento, tan solo debemos asociar un manejador y capturar los datos de salida con `$event`.

```
@Component({ selector: "posts" })
export class PostsComponent {

 private _page: number = 0;
 @Output() pageChanged: EventEmitter<number> = new EventEmitter();

 onNextPageButtonClick() { this.pageChanged.emit(++this._page); };
}

@Component({ template: `<posts (pageChanged)="onPage($event)"></posts>` })
export class AppComponent { ... }
```



Angular - Components

dudas & preguntas

Services

Angular - Services

- Definen bloques de código reusables.
- Son singletons.
- Necesitan inyectarse como dependencias.

Angular - Services

Convención

Documento	`user-config.service.ts`
Clase	`UserConfigService`

Importación

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

Angular - Services

```
@Injectable()
export class ConfigService {

  getConfig() {
    return {
      language: "en-US",
      paginationItems: 20
    };
  }
}
```

Angular - Services

```
@Injectable()
export class ConfigService {

  getConfig() {
    return {
      language: "en-US",
      paginationItems: 20
    };
  }
}
```

El decorador `@Injectable` no espera metadatos. Tan solo decora una clase para hacer que pueda inyectarse como una dependencia.

Angular - Services

```
@NgModule({
  ...
  providers: [ConfigService]
  ...
})
export class AppModule { }
```

Angular - Services

Es necesario añadir el servicio al metadato `providers` del módulo. En caso de no hacerlo, Angular no sabrá cómo inyectarlo como dependencia cuando se solicite.

```
@NgModule({
  ...
  providers: [ConfigService]
  ...
})
export class AppModule { }
```

Angular - Services

```
@Component({...})
export class UserComponent {

 constructor(private _configService: ConfigService) { }
}
```

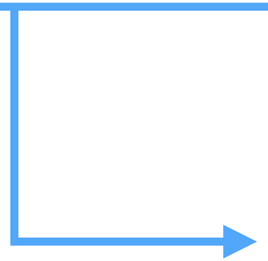
Angular - Services

Indicando el servicio como parámetro del constructor del componente, hacemos que Angular nos inyecte una instancia del mismo.

```
@Component({...})
export class UserComponent {

 constructor(private _configService: ConfigService) { }
}
```

Angular - Services

```
@Component({...})
export class UserComponent {

  constructor(private _configService: ConfigService) { }
}
```



La instancia del servicio se almacena en una variable privada del componente.

Angular - Services

dudas & preguntas

Dependency Injection

Angular - Dependency Injection

- Es un patrón de diseño.
- Las dependencias de una clase se suministran ya resueltas.
- No es responsabilidad de una clase instanciar sus dependencias.
- Angular Injector se encarga de hacer todo el trabajo necesario.

Angular - Dependency Injection

Ejemplo sin usar inyección de dependencias.

Angular - Dependency Injection

```
@Component({...})
export class ProductComponent {

  private _productService: ProductService;

  constructor() {
    this._productService = new ProductService();
  }
}
```

Angular - Dependency Injection

```
@Component({...})
export class ProductComponent {

  private _productService: ProductService;

  constructor() {
    this._productService = new ProductService()
  }
}
```

No debería ser responsabilidad de la clase instanciar sus dependencias.

Angular - Dependency Injection

Ejemplo usando inyección de dependencias.

Angular - Dependency Injection

```
@Component({...})
export class ProductComponent {

 constructor(@Inject(ProductService) private _productService) { }
}
```

Angular - Dependency Injection

```
@Component({...})
export class ProductComponent {

 constructor(@Inject(ProductService) private _productService) { }
}
```

Cuando decoramos con `@Inject` un parámetro del constructor de una clase, Angular Injector resuelve la dependencia y la entrega ya instanciada.

Angular - Dependency Injection

```
@Component({...})
export class ProductComponent {

  constructor(@Inject(ProductService) private _productService) { }
}
```

`@Inject` está en el paquete `@angular/core`.

Angular - Dependency Injection

Inyección de dependencias simplificada.

Angular - Dependency Injection

```
@Component({...})
export class ProductComponent {

 constructor(private _productService: ProductService) { }
}
```

Angular - Dependency Injection

```
@Component({...})
export class ProductComponent {

 constructor(private _productService: ProductService) { }
}
```

Si `@Inject` no está presente, Angular Injector usa la anotación de tipo.

Angular - Dependency Injection

Es necesario registrar los proveedores de los servicios.

Angular - Dependency Injection

```
@NgModule({
  ...
  providers: [ProductService]
  ...
})
export class AppModule { }
```

Angular - Dependency Injection

Si no se registra el proveedor del servicio, Angular Injector no sabrá cómo instanciar dicho servicio para inyectarlo como dependencia.

Angular - Dependency Injection

El registro de proveedores puede personalizarse.

Angular - Dependency Injection

Proveedor de clase.

Angular - Dependency Injection

```
@NgModule({
  providers: [{
    provide: ProductService,
    useClass: ProductService
  }]
})
export class AppModule { }
```

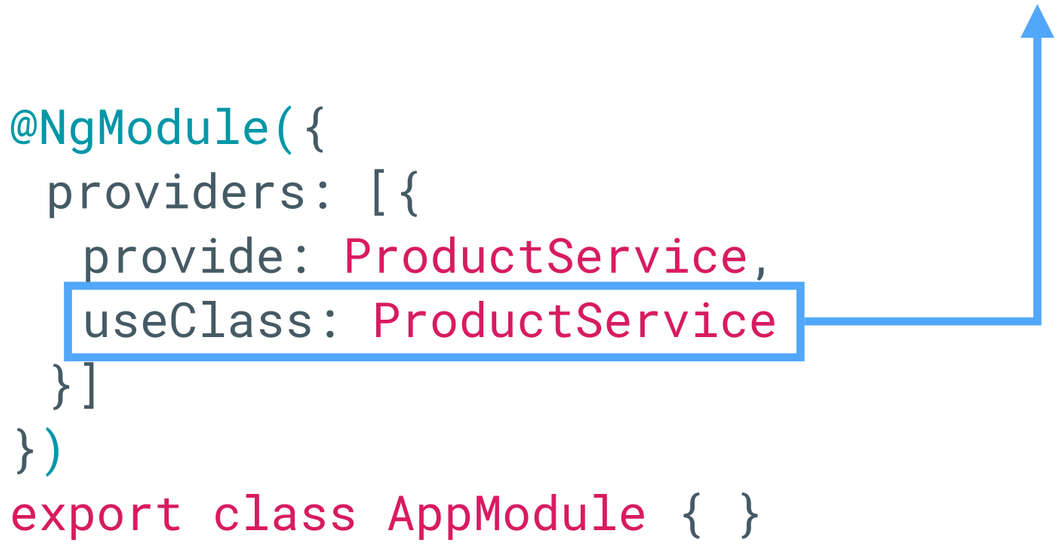
Angular - Dependency Injection

Con `provide` indicamos un token. Este token es el que usaremos para solicitar una inyección de dependencia.

```
@NgModule({
  providers: [{
    provide: ProductService
    useClass: ProductService
  }]
})
export class AppModule { }
```

Angular - Dependency Injection

Con `useClass` indicamos la clase a instanciar cuando se solicite una inyección de dependencia de este tipo.



```
@NgModule({
  providers: [{
    provide: ProductService,
    useClass: ProductService
  }]
})
export class AppModule { }
```

Angular - Dependency Injection

Proveedor de factoría.

Angular - Dependency Injection

```
@NgModule({
  providers: [{
    provide: LogService,
    useFactory: () => {
      return new LogService("LEVEL_INFO");
    }
  }]
})
export class AppModule { }
```

Angular - Dependency Injection

Con `useFactory` indicamos una función que resuelva la instanciación del servicio en cuestión.

```
@NgModule({
  providers: [{
    provide: LogService,
    useFactory: () => {
      return new LogService("LEVEL_INFO");
    }
  }]
})
export class AppModule { }
```

Angular - Dependency Injection

```
@NgModule({
 providers: [{
  provide: LogService,
  deps: [Http, Console],
  useFactory: (http, console) => {
   return new LogService(http, console);
  }
 }]
})
export class AppModule { }
```

Angular - Dependency Injection

En deps indicamos las dependencias que tenga, a su vez, el servicio cuyo proveedor estamos registrando.

```
@NgModule({
 providers: [{
  provide: LogService,
  deps: [Http, Console]
  useFactory: (http, console) => {
   return new LogService(http, console);
  }
 }]
})
export class AppModule { }
```

Angular - Dependency Injection

```
@NgModule({
 providers: [{
  provide: LogService,
  deps: [Http, Console],
  useFactory: (http, console) => {
   return new LogService(http, console);
  }
 }]
})
export class AppModule { }
```

Todas las dependencias indicadas en `deps` se resuelven y se pasan como parámetros a la factoría respetando el orden.

Angular - Dependency Injection

Proveedor de valor.

Angular - Dependency Injection

```
@NgModule({
 providers: [{
  provide: "MyMessage",
  useValue: "I'am an injected message"
 }]
})
export class AppModule { }
```

Angular - Dependency Injection

```
@NgModule({
  providers: [{
    provide: "MyMessage",
    useValue: "I'am an injected message"
  }]
})
export class AppModule { }
```

Con `useValue` indicamos el tipo primitivo a inyectar cuando se solicite una dependencia de este tipo.

Angular - Dependency Injection

No se recomienda usar una cadena de texto como token.

```
@NgModule({
  providers: [{
    provide: "MyMessage"
    useValue: "I'am an injected message"
  }]
})
export class AppModule { }
```

Angular - Dependency Injection

```
const MyMessage = new OpaqueToken("MyMessage");

@NgModule({
 providers: [{
  provide: MyMessage,
  useValue: "I'am an injected message"
 }]
})
export class AppModule { }
```

Angular - Dependency Injection

```
const MyMessage = new OpaqueToken("MyMessage");

@NgModule({
  providers: [{
    provide: MyMessage
    useValue: "I'am an injected message"
  }]
})
export class AppModule { }
```

Creamos un `OpaqueToken` y lo usamos para registrar el proveedor.

Angular - Dependency Injection

```
const MyMessage = new OpaqueToken("MyMessage");

@NgModule({
  providers: [{
    provide: MyMessage,
    useValue: "I'am an injected message"
  }]
})
export class AppModule { }
```

`OpaqueToken` está en el paquete `@angular/core`.

Angular - Dependency Injection

dudas & preguntas

Forms

Angular - Forms

Clases `FormGroup` y `FormControl` al rescate.

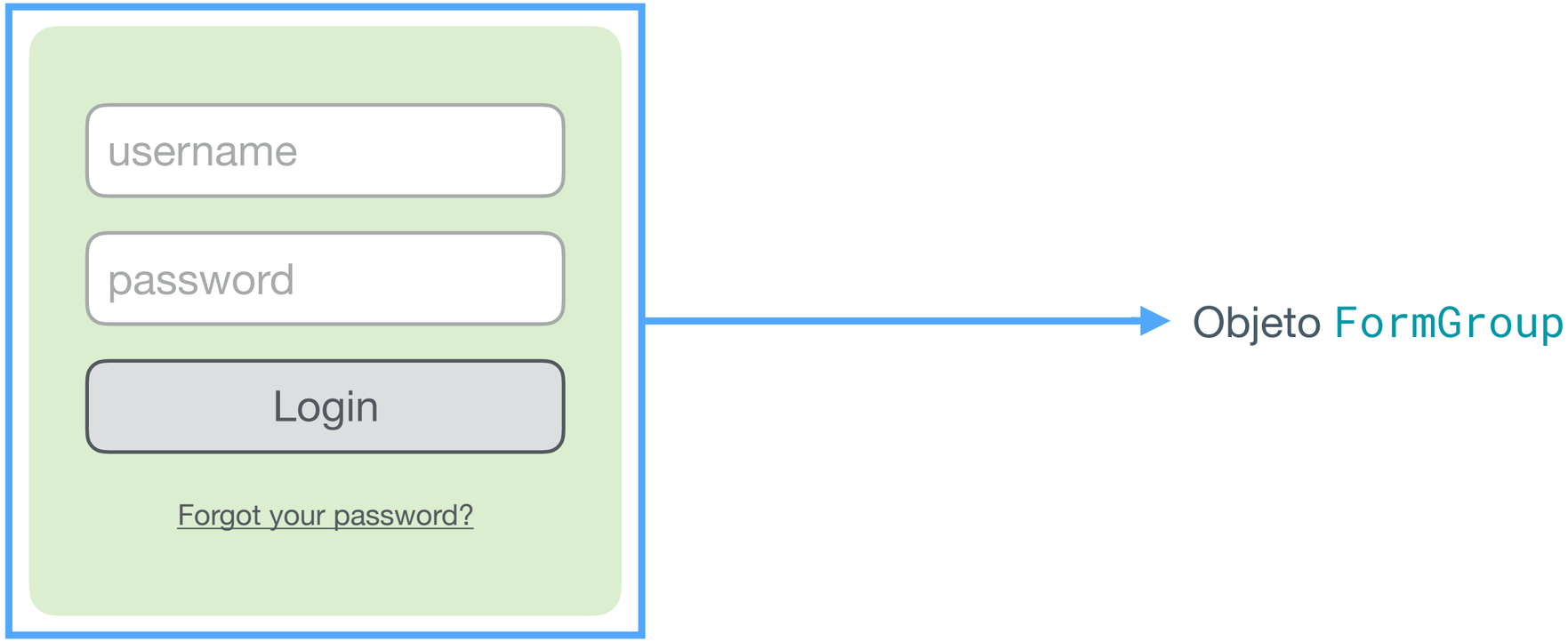
Angular - Forms

Angular - Forms

Clase `FormGroup`:

- Añade *vitaminas* a un elemento formulario.
- Se instancia automáticamente para cada formulario.
- Gestiona los objetos `FormControl` que contiene.
- Se enlaza al elemento a través de la directiva `NgForm`.

Angular - Forms

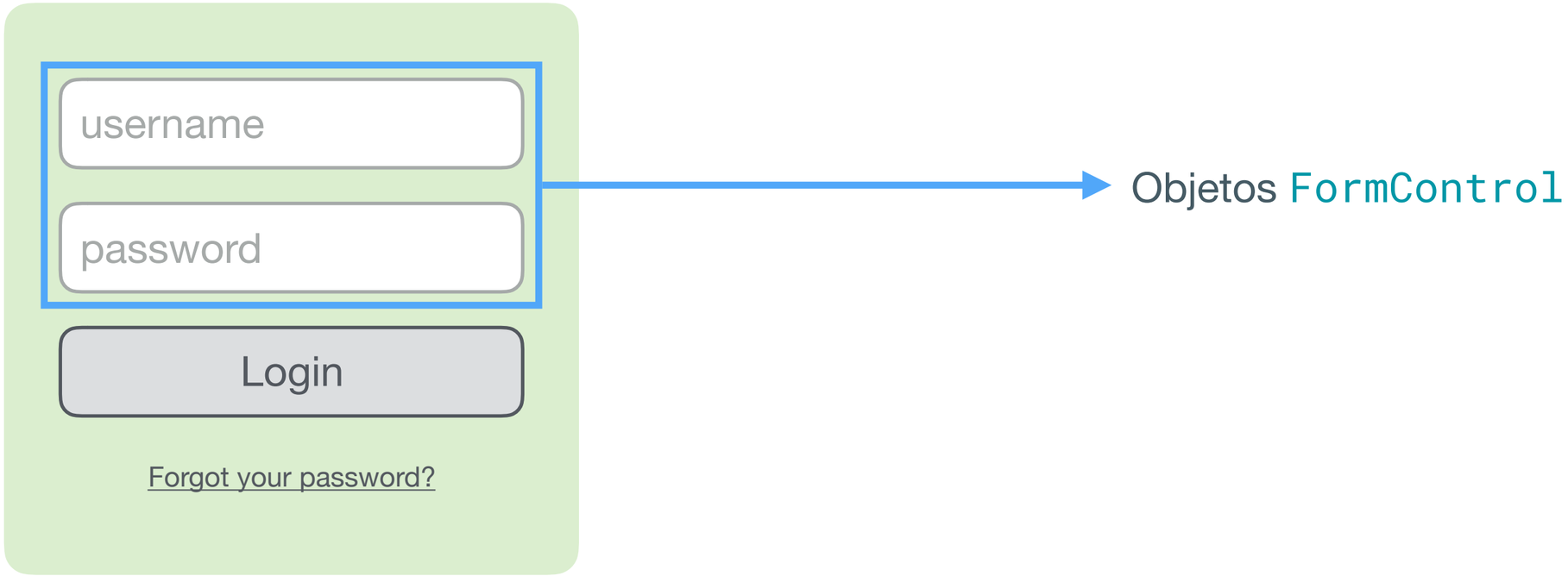


Angular - Forms

Clase `FormControl`:

- Añade *vitaminas* a un elemento de formulario.
- Hace seguimiento del valor y estado del elemento.
- Se enlaza al elemento a través de la directiva `NgModel`.

Angular - Forms



Angular - Forms

¿Qué necesitamos para trabajar con formularios?

Angular - Forms

1. Importar `FormsModule` en nuestro módulo

Angular - Forms

```
import { FormsModule } from "@angular/forms";

@NgModule({
  ...
  import: [FormsModule]
  ...
})
export class AppModule { }
```

Angular - Forms

```
import { FormsModule } from "@angular/forms";

@NgModule({
  ...
  import: [FormsModule]
  ...
})
export class AppModule { }
```

El módulo `FormsModule` está en la librería `@angular/forms`.

Angular - Forms

```
import { FormsModule } from "@angular/forms";

@NgModule({
  ...
  import: [FormsModule]
  ...
})
export class AppModule { }
```

Lo añadimos al metadato `import` de nuestro módulo.

Angular - Forms

2. Go! Go! Go!

Angular - Forms

Trabajando con `FormGroup` a través de `NgForm`.

Angular - Forms

```
<form #login="ngForm" (ngSubmit)="onSubmit(login)">
  ...
  <button type="submit" [disabled]="login.invalid">Login</button>
</form>
```

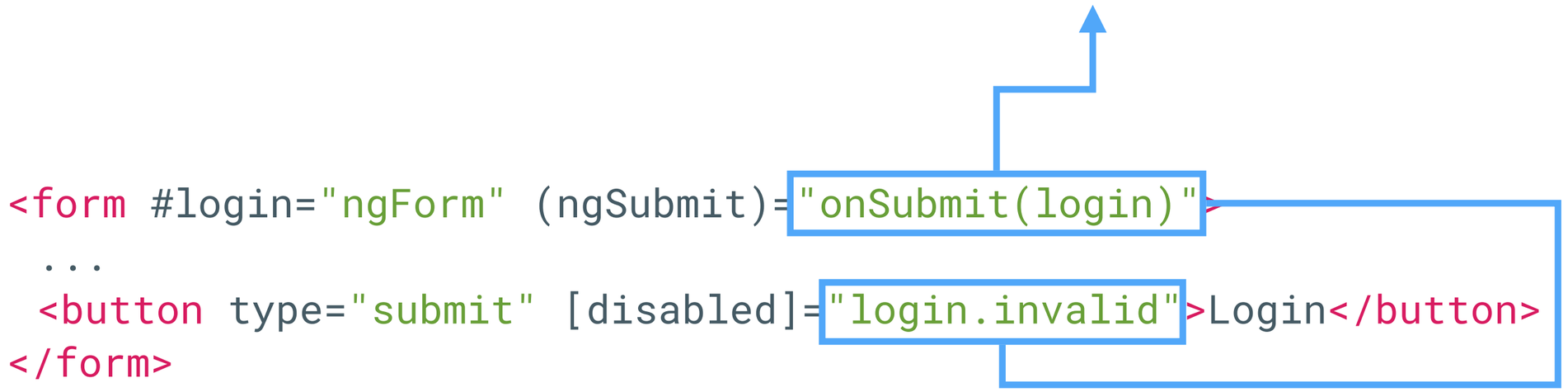
Angular - Forms

```
<form #login="ngForm" (ngSubmit)="onSubmit(login)">
  ...
  <button type="submit" [disabled]="login.invalid">Login</button>
</form>
```

Creamos una ***template reference variable*** usando la notación # y en ella asignamos el `FormGroup` correspondiente al formulario, que viene dado a través de la directiva `NgForm`.

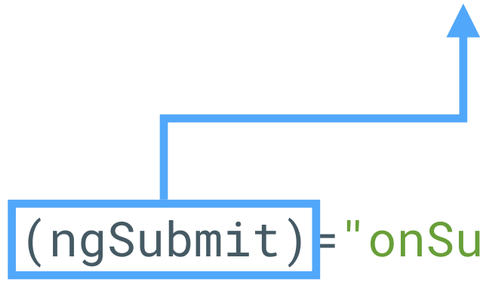
Angular - Forms

Podemos acceder a todas las propiedades del formulario a través de la ***template reference variable***.



Angular - Forms

`ngSubmit` es el evento que se dispara cuando se envía el formulario. Esta funcionalidad viene dada por `NgForm`.



```
<form #login="ngForm" (ngSubmit)="onSubmit(login)">
  ...
  <button type="submit" [disabled]="login.invalid">Login</button>
</form>
```

Angular - Forms

Trabajando con `FormControl` a través de `NgModel`.

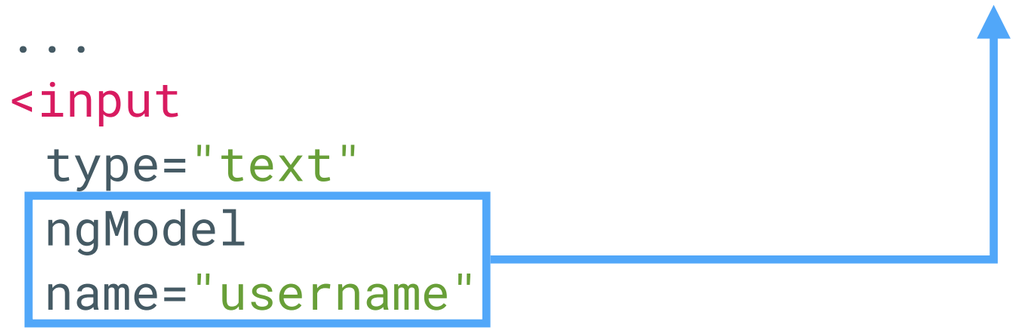
Angular - Forms

```
<form>
  ...
  <input
    type="text"
    ngModel
    name="username"
    #user="ngModel"
    required />
  <div [hidden]="user.valid">
    The username is a required field.
  </div>
  ...
</form>
```

Angular - Forms

Con `ngModel` instanciamos un objeto `FormControl`. La propiedad `name` del elemento debe estar definida.

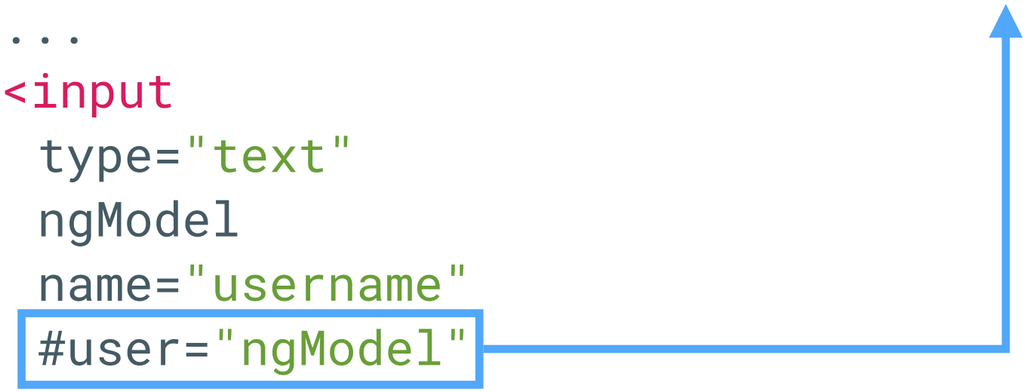
```
<form>
  ...
  <input
    type="text"
    ngModel
    name="username"
    #user="ngModel"
    required />
  <div [hidden]="user.valid">
    The username is a required field.
  </div>
  ...
</form>
```



Angular - Forms

Creamos una ***template reference variable*** usando la notación `#` y en ella asignamos el `FormControl` correspondiente al elemento, que viene dado a través de la directiva `NgModel`.

```
<form>
  ...
  <input
    type="text"
    ngModel
    name="username"
    #user="ngModel"
    required />
  <div [hidden]="user.valid">
    The username is a required field.
  </div>
  ...
</form>
```



Angular - Forms

Podemos acceder a todas las propiedades del elemento a través de la ***template reference variable***.

```
<form>
  ...
  <input
    type="text"
    ngModel
    name="username"
    #user="ngModel"
    required />
  <div [hidden]="user.valid">
    The username is a required field.
  </div>
  ...
</form>
```

Angular - Forms

Validación del formulario: estilizando los elementos.

Angular - Forms

Clases CSS del elemento

	Sí	No
Si el control se ha visitado	`ng-touched`	`ng-untouched`
Si el valor del control ha cambiado	`ng-dirty`	`ng-pristine`
Si el valor del control es válido	`ng-valid`	`ng-invalid`

Angular - Forms

```
input.ng-touched.ng-invalid {
  border: 1px solid red;
}

input.ng-touched.ng-valid {
  border: 1px solid green;
}
```

Angular - Forms

Si el control se ha visitado pero su valor no es válido.

```
input.ng-touched.ng-invalid {
  border: 1px solid red;
}

input.ng-touched.ng-valid {
  border: 1px solid green;
}
```

Angular - Forms

```
input.ng-touched.ng-invalid {
  border: 1px solid red;
}

input.ng-touched.ng-valid {
  border: 1px solid green;
}
```

Si el control se ha visitado y su valor es válido.

Angular - Forms

Validación del formulario: mostrando mensajes.

Angular - Forms

```
<input
 type="password"
 ngModel
 name="password"
 #pass="ngModel"
 required />
<div *ngIf="pass.touched && pass.invalid">
 The password is required.
</div>
```

Angular - Forms

```
<input
 type="password"
 ngModel
 name="password"
 #pass="ngModel"
 required />
<div *ngIf="pass.touched && pass.invalid">
 The password is required.
</div>
```

Si el control se ha visitado pero su valor no es válido, se muestra el mensaje de error correspondiente.

Angular - Forms

dudas & preguntas

Routing

Angular - Routing

- Aporta Single Page Application.
- La navegación se simula con HTML5 History API.
- Angular se encarga de hacer la manipulación del DOM.
- Mejora la experiencia de usuario.

Angular - Routing

1. Establecer / como URL base

Angular - Routing

```
<!DOCTYPE html>
<html>
 <head>
  <base href="/">
  ...
 </head>
 <body>
  ...
 </body>
</html>
```

Angular - Routing

```
<!DOCTYPE html>
<html>
  <head>
    <base href="/">
    ...
  </head>
  <body>
    ...
  </body>
</html>
```

El elemento `base` debe establecerse antes de referenciar cualquier documento, como hojas de estilo CSS, favicon o scripts.

Angular - Routing

2. Configurar las rutas en un Routing Module

Angular - Routing

Convención

Documento	`app-routing.module.ts`
Clase	`AppRoutingModule`

Importación

```
import { RouterModule } from "@angular/router";
```

Angular - Routing

```
const routes = [{
 path: "",
 component: HomeComponent
}, {
 path: "admin",
 component: AdminComponent
}];

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

Angular - Routing

```
const routes = [{
 path: "",
 component: HomeComponent
}, {
 path: "admin",
 component: AdminComponent
}];
```

Creamos tantas rutas como sean necesarias. Cada ruta es un par `path` / `component`.

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

Angular - Routing

```
const routes = [{
 path: ""
 component: HomeComponent
}, {
 path: "admin",
 component: AdminComponent
}];

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

En la propiedad `path` indicamos la ruta que se navegará.

Angular - Routing

```
const routes = [{
 path: "",
 component: HomeComponent
}, {
 path: "admin",
 component: AdminComponent
}];

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

En la propiedad `component` indicamos el componente encargado de responder a la ruta navegada.

Angular - Routing

```
const routes = [{
 path: "",
 component: HomeComponent
}, {
 path: "admin",
 component: AdminComponent
}];

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

Con la función estática `forRoot()` creamos un módulo que incluye todo lo necesario (router y directivas) para gestionar las rutas definidas.

Angular - Routing

```
const routes = [{
 path: "",
 component: HomeComponent
}, {
 path: "admin",
 component: AdminComponent
}];

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

Exportamos el módulo `RouterModule` para que nuestro módulo pueda hacer uso de todas las funciones que expone.

Angular - Routing

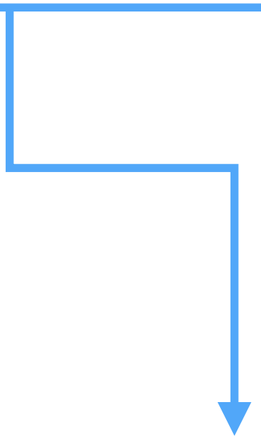
3. Importar el Router Module en nuestro módulo

Angular - Routing

```
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  ...
  import: [AppRoutingModule]
  ...
})
export class AppModule { }
```

Angular - Routing

```
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  ...
  import: [AppRoutingModule]
  ...
})
export class AppModule { }
```



Importamos `AppRoutingModule` del documento que hemos creado previamente.

Angular - Routing

```
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  ...
  import: [AppRoutingModule]
  ...
})
export class AppModule { }
```

Lo añadimos al metadato `import` de nuestro módulo.

Angular - Routing

4. Añadir una directiva `RouterOutlet`

Angular - Routing

```
@Component({
  template: `
    ...
    <router-outlet></router-outlet>
  `
})
export class AppComponent { }
```

Angular - Routing

La directiva `RouterOutlet` proporciona el contenedor donde se instanciarán los componentes encargados de manejar las distintas rutas que se naveguen.

```
@Component({
  template: `
    ...
    <router-outlet></router-outlet>
  `
})
export class AppComponent { }
```

Angular - Routing

5. Añadir enlaces con la directiva `routerLink`

Angular - Routing

```
@Component({
  template: `
    <nav>
      <a routerLink="/">Home</a>
      <a routerLink="/admin">Admin</a>
    </nav>
    ...
  `
})
export class AppComponent { }
```

Angular - Routing

Hacemos uso de `routerLink` para establecer los enlaces de las distintas secciones.

```
@Component({
  template: `
    <nav>
      <a routerLink="/">Home</a>
      <a routerLink="/admin">Admin</a>
    </nav>
    ...
  `
})
export class AppComponent { }
```

Angular - Routing

6. Yippee-ki-yay!

Angular - Routing

Trabajando con rutas parametrizadas.

Angular - Routing

```
const routes = [{
  path: "users/:userId",
  component: UserDetailsComponent
}];
```

Angular - Routing

```
const routes = [{
  path: "users/:userId"
  component: UserDetailsComponent
}];
```

Podemos definir parámetros de ruta precediéndolos con : en la definición de la misma.

Angular - Routing

```
@Component({
 template: `
  <ul>
   <li *ngFor="let user of users">
    <a [routerLink]="['users', user.id]">{{ user.name }}</a>
   <li>
  </ul>
 `
})
export class UsersComponent { }
```

Angular - Routing

Envolvemos la directiva `RouterLink` con `[]` para indicarle que evalúe la expresión asignada.

```
@Component({
  template: `
    <ul>
      <li *ngFor="let user of users">
        <a [routerLink]="['users', user.id]">{{ user.name }}</a>
      <li>
    </ul>
  `
})
export class UsersComponent { }
```

Angular - Routing

```
@Component({
  template: `
    <ul>
      <li *ngFor="let user of users">
        <a [routerLink]="['users', user.id]">{{ user.name }}</a>
      <li>
    </ul>
  `
})
export class UsersComponent { }
```

Podemos establecer un array como expresión. En este caso, indicamos como primer elemento la parte estática de la ruta. Y como segundo, el identificador de la entidad.

Angular - Routing

```
@Component({ template: "..." })
export class UserDetailsComponent implements OnInit {

 constructor(private _route: ActivatedRoute) { }

 ngOnInit() {
  this._route.params.forEach((param: { userId: string }) => {
   /* Do something with param.userId */
  });
 }
}
```

Angular - Routing

```
@Component({ template: "..." })
export class UserDetailsComponent implements OnInit {

 constructor(private _route: ActivatedRoute) { }

  ngOnInit() {
    this._route.params.forEach((param: { userId: string }) => {
      /* Do something with param.userId */
    });
  }
}
```

Hacemos la inyección de dependencia de `ActivatedRoute`.

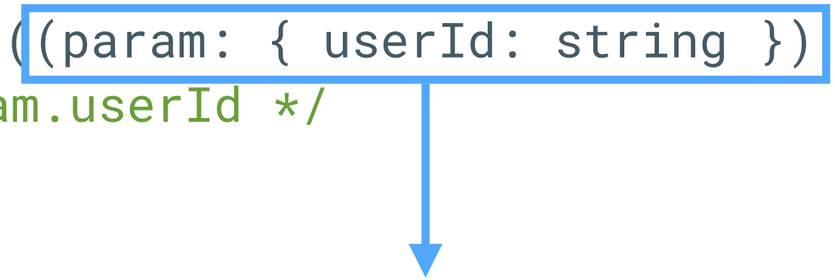
Angular - Routing

```
@Component({ template: "..." })
export class UserDetailsComponent implements OnInit {

 constructor(private _route: ActivatedRoute) { }

 ngOnInit() {
  this._route.params forEach((param: { userId: string }) => {
    /* Do something with param.userId */
  });
 }
}
```

El objeto `params` es un observable que tiene toda la información relacionada con los parámetros de ruta.

Angular - Routing

```
@Component({ template: "..." })
export class UserDetailsComponent implements OnInit {

 constructor(private _route: ActivatedRoute) { }

 ngOnInit() {
  this._route.params.forEach((param: { userId: string }) => {
   /* Do something with param.userId */
  });
 }
}
```



Podemos obtener el parámetro de ruta `userId` a través de la función `forEach` del objeto observable.

Angular - Routing

Podemos resolver datos en la navegación de una ruta.

Angular - Routing

```
const routes = [{
  path: "users/:userId",
  component: UserDetailsComponent,
  resolve: {
    user: UserDetailsResolve
  }
}];
```

Angular - Routing

```
const routes = [{
  path: "users/:userId",
  component: UserDetailsComponent,
  resolve: {
    user: UserDetailsResolve
  }
}];
```

En la definición de la ruta, hacemos uso de la propiedad `resolve`. Podemos resolver tantos datos como necesitemos en el componente que se navega.

Angular - Routing

```
const routes = [{
  path: "users/:userId",
  component: UserDetailsComponent,
  resolve: {
    user: UserDetailsResolve
  }
}];
```

Para resolver los datos nos apoyamos en un servicio que implementa la interfaz `Resolve`.

Angular - Routing

Convención

Documento	`user-details-resolve.service.ts`
Clase	`UserDetailsResolve`

Importación

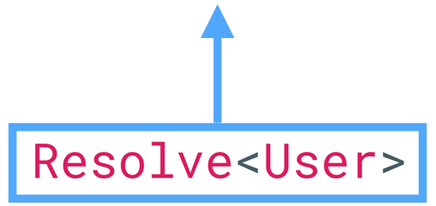
```
import { ActivatedRouteSnapshot, Resolve } from "@angular/router";
```

Angular - Routing

```
@Injectable()
export class UserDetailsResolve implements Resolve<User> {

 constructor(private _backend: BackendService) { }

 resolve(route: ActivatedRouteSnapshot): Observable<User> {
  return this._backend.getUser(route.params.userId);
 }
}
```

Angular - Routing

Implementamos la interfaz `Resolve` indicando el tipo de dato a resolver.



```
@Injectable()
export class UserDetailsResolve implements Resolve<User> {

 constructor(private _backend: BackendService) { }

 resolve(route: ActivatedRouteSnapshot): Observable<User> {
  return this._backend.getUser(route.params.userId);
 }
}
```

Angular - Routing

```
@Injectable()
export class UserDetailsResolve implements Resolve<User> {

 constructor(private _backend: BackendService) { }

 resolve(route: ActivatedRouteSnapshot): Observable<User> {
  return this._backend.getUser(route.params.userId);
 }
}
```

La función `resolve` nos viene impuesta por la interfaz.

Angular - Routing

```
@Injectable()
export class UserDetailsResolve implements Resolve<User> {

 constructor(private _backend: BackendService) { }

 resolve(route: ActivatedRouteSnapshot): Observable<User> {
  return this._backend.getUser(route.params.userId);
 }
}
```

Tenemos acceso a la ruta que se está navegando, y por tanto a los parámetros que haya definidos en ella.

Angular - Routing

```
@Injectable()
export class UserDetailsResolve implements Resolve<User> {

 constructor(private _backend: BackendService) { }

 resolve(route: ActivatedRouteSnapshot): Observable<User> {
  return this._backend.getUser(route.params.userId);
 }
}
```

Podemos retornar observables o promesas si queremos trabajar con datos asíncronos.

Angular - Routing

```
@Component({ template: "..." })
export class UserDetailsComponent implements OnInit {

 constructor(private _route: ActivatedRoute) { }

 ngOnInit() {
  this._route.data.forEach((data: { user: User }) => {
   /* Do something with data.user */
  });
 }
}
```

Angular - Routing

```
@Component({ template: "..." })
export class UserDetailsComponent implements OnInit {

 constructor(private _route: ActivatedRoute) { }

 ngOnInit() {
  this._route.data.forEach((data: { user: User }) => {
   /* Do something with data.user */
  });
 }
}
```

Para obtener un dato resuelto, buscamos en la propiedad `data` de la ruta.

Angular - Routing

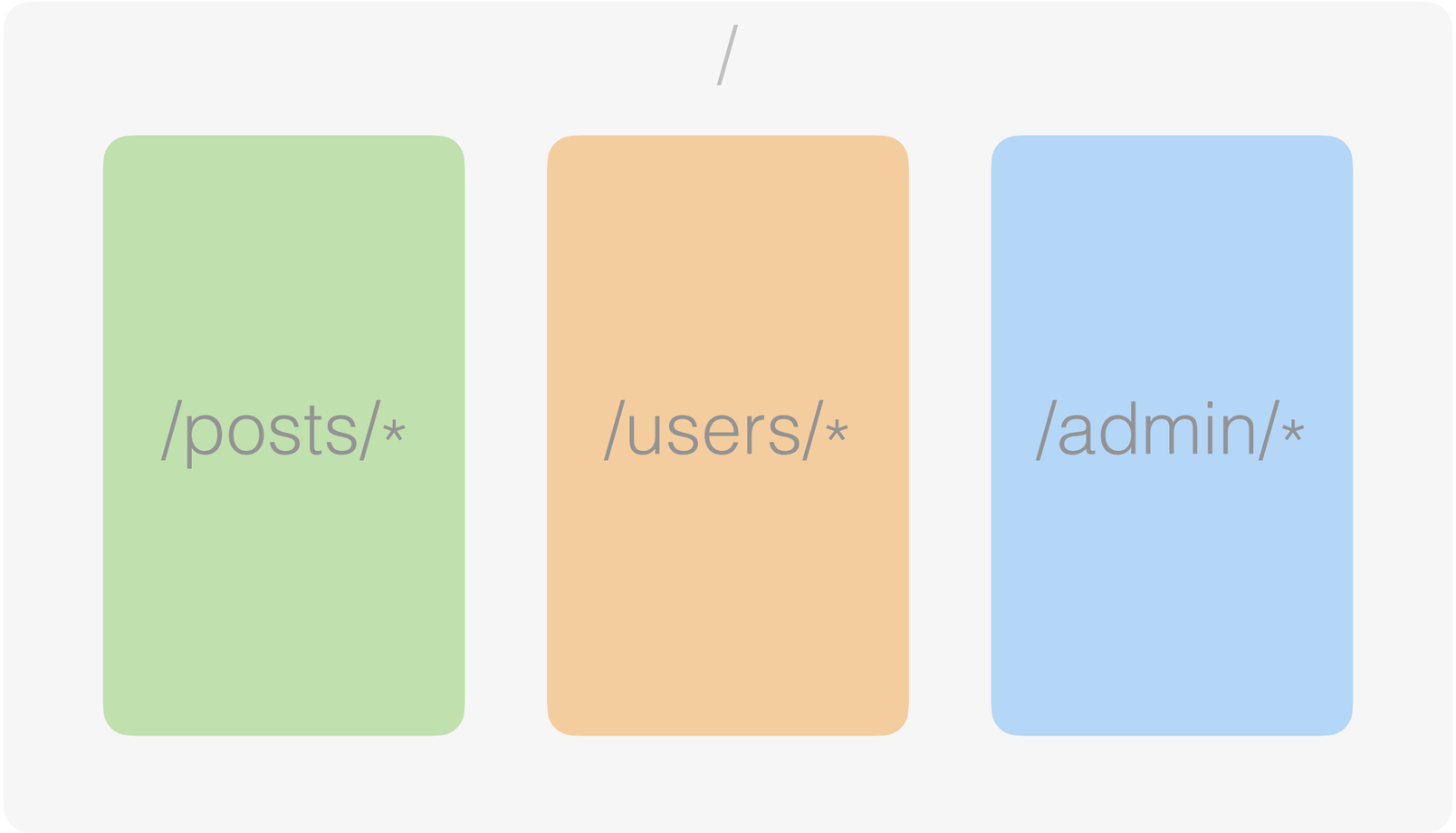
```
@Component({ template: "..." })
export class UserDetailsComponent implements OnInit {

 constructor(private _route: ActivatedRoute) { }

 ngOnInit() {
  this._route.data.forEach((data: { user: User }) => {
   /* Do something with data.user */
  });
 }
}
```

Buscamos el dato por el nombre que usamos en la definición de la ruta en la propiedad `resolve`.

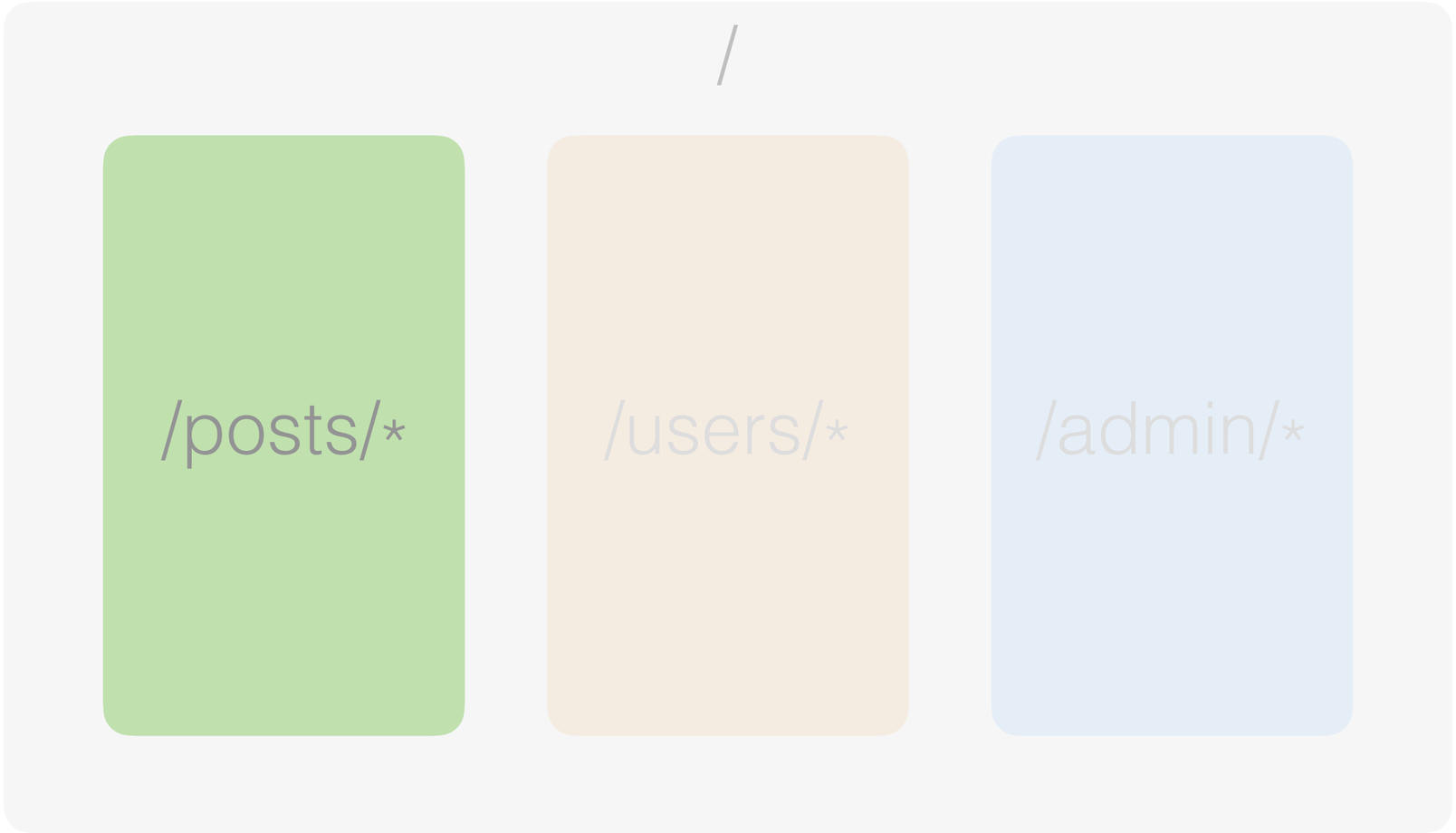
Angular - Routing

Cargando rutas de manera perezosa.

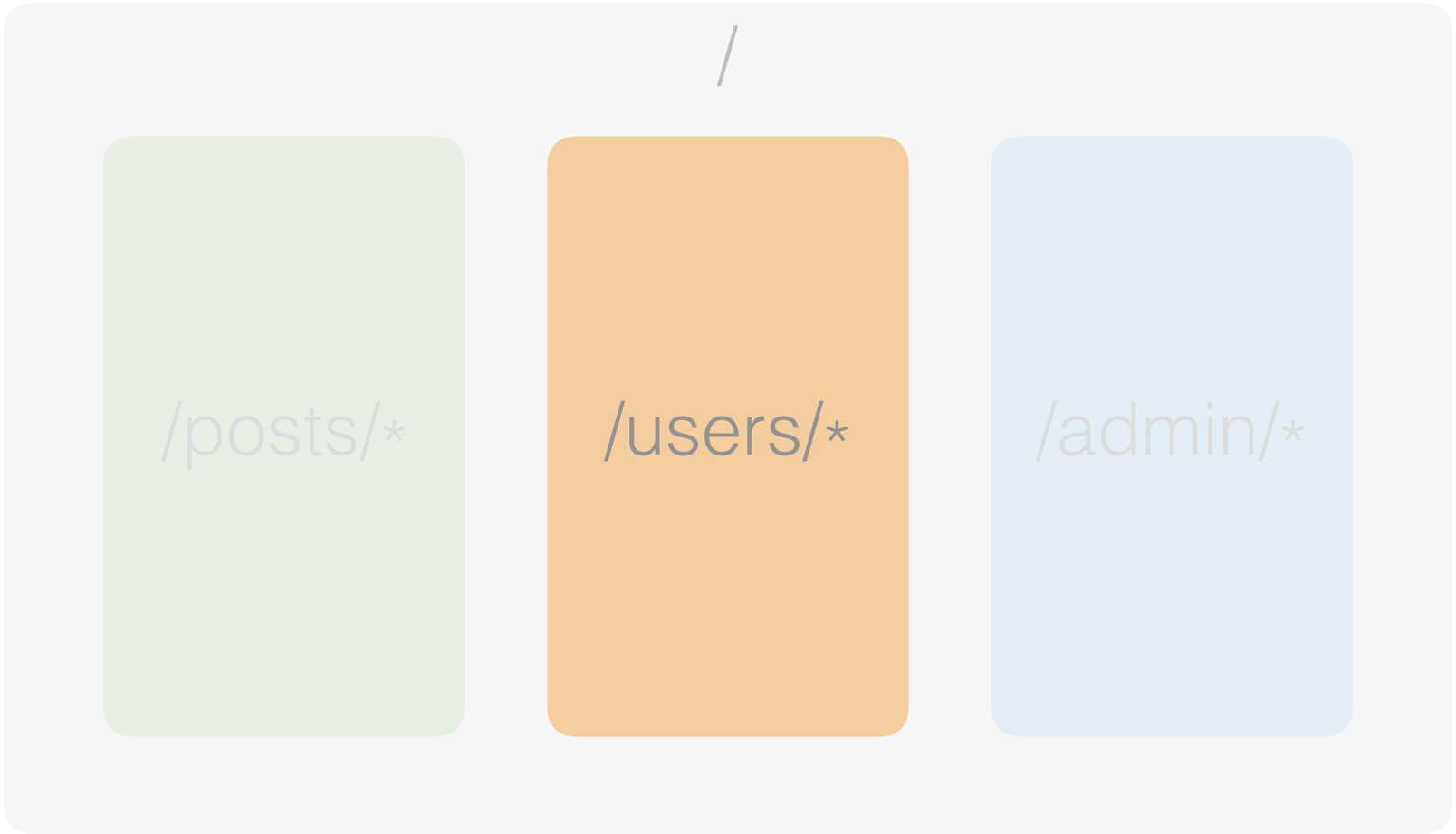
Angular - Routing



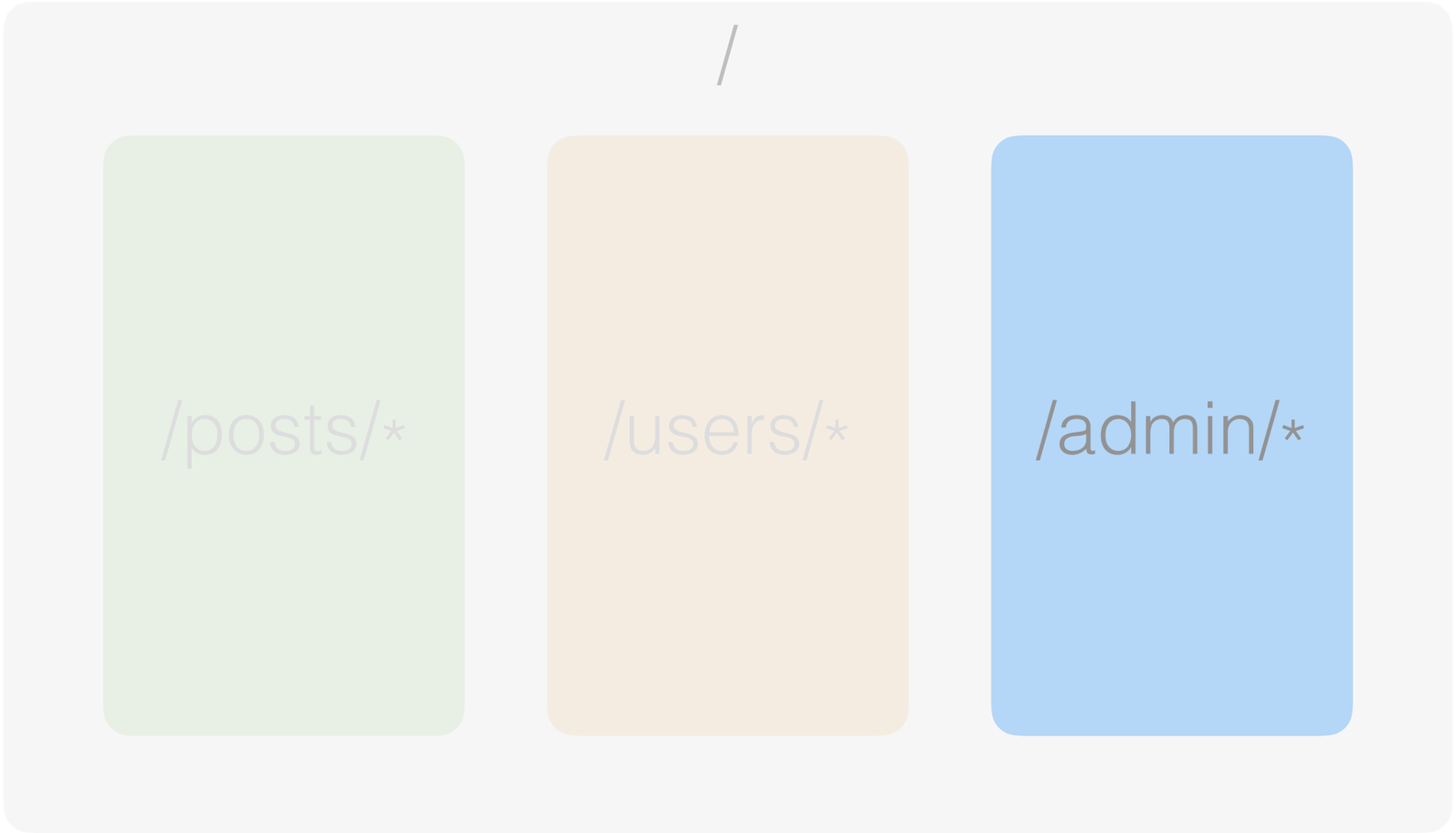
Angular - Routing



Angular - Routing



Angular - Routing



Angular - Routing

1. Creamos un Routing Module para cada sección

Angular - Routing

```
const postsRoutes = [...];

@NgModule({
  imports: [RouterModule.forChild(postsRoutes)],
  exports: [RouterModule]
})
export class PostsRoutingModule { }

...
export class UsersRoutingModule { }

...
export class AdminRoutingModule { }
```

Angular - Routing

```
const postsRoutes = [...];

@NgModule({
  imports: [RouterModule.forChild(postsRoutes)],
  exports: [RouterModule]
})
export class PostsRoutingModule { }
```

```
...
export class UsersRoutingModule { }
```

```
...
export class AdminRoutingModule { }
```

Angular - Routing

```
const postsRoutes = [...];

@NgModule({
  imports: [RouterModule.forChild(postsRoutes)],
  exports: [RouterModule]
})
export class PostsRoutingModule { }

...
export class UsersRoutingModule { }

...
export class AdminRoutingModule { }
```

Esta vez usamos la función estática `forChild`, que incluye todo lo necesario a excepción del servicio `Router`.

Angular - Routing

2. Registrar las rutas en el Router Module principal

Angular - Routing

```
const routes = [{
 path: "posts",
 loadChildren: "app/posts/posts.routing#PostsRoutingModule"
}, {
 path: "users",
 loadChildren: "app/users/users.routing#UsersRoutingModule"
}, {
 path: "admin",
 loadChildren: "app/admin/admin.routing#AdminRoutingModule"
}];
```

Angular - Routing

En la propiedad `loadChildren` establecemos la colección de rutas hijas que cuelgan de `path`.

```
const routes = [{
  path: "posts",
  loadChildren: "app/posts/posts.routing#PostsRoutingModule"
}, {
  path: "users",
  loadChildren: "app/users/users.routing#UsersRoutingModule"
}, {
  path: "admin",
  loadChildren: "app/admin/admin.routing#AdminRoutingModule"
}];
```

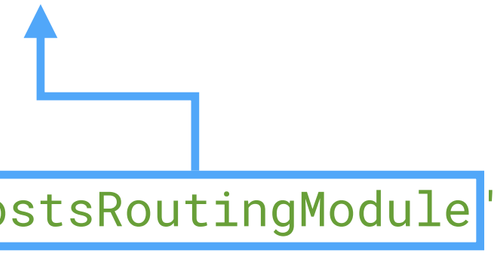
Angular - Routing

Indicamos la ruta del Router Module correspondiente.

```
const routes = [{
 path: "posts",
 loadChildren: "app/posts/posts.routing#PostsRoutingModule"
}, {
 path: "users",
 loadChildren: "app/users/users.routing#UsersRoutingModule"
}, {
 path: "admin",
 loadChildren: "app/admin/admin.routing#AdminRoutingModule"
}];
```

Angular - Routing

En el documento indicado puede haber varias exportaciones. Con # indicamos cuál queremos usar.



```
const routes = [{
 path: "posts",
 loadChildren: "app/posts/posts.routing#PostsRoutingModule"
}, {
 path: "users",
 loadChildren: "app/users/users.routing#UsersRoutingModule"
}, {
 path: "admin",
 loadChildren: "app/admin/admin.routing#AdminRoutingModule"
}];
```

Angular - Routing

dudas & preguntas

HTTP Client

Angular - HTTP Client

- Facilita el trabajo con peticiones HTTP.
- Se apoya en objetos observables.

Angular - HTTP Client

1. Importar `HttpModule` en nuestro módulo

Angular - HTTP Client

```
import { HttpModule } from "@angular/http";

@NgModule({
  ...
  import: [HttpModule]
  ...
})
export class AppModule { }
```

Angular - HTTP Client

```
import { HttpModule } from "@angular/http";

@NgModule({
  ...
  import: [HttpModule]
  ...
})
export class AppModule { }
```

El módulo `HttpModule` está en la librería `@angular/http`.

Angular - HTTP Client

```
import { HttpModule } from "@angular/http";

@NgModule({
  ...
  import: [HttpModule]
  ...
})
export class AppModule { }
```

Lo añadimos al metadato `import` de nuestro módulo.

Angular - HTTP Client

2. Inyectar el cliente `Http` como dependencia

Angular - HTTP Client

```
import { Http } from "@angular/http";

@Injectable()
export class ProductService {

 constructor(private _http: Http) { }
}
```

Angular - HTTP Client

```
import { Http } from "@angular/http";

@Injectable()
export class ProductService {

  constructor(private _http: Http) { }
}
```

Hacemos la inyección de dependencia de `Http`.

Angular - HTTP Client

3. Importar la clase y operadores de `Observable`

Angular - HTTP Client

```
import { Response } from "@angular/http";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/map";
```

Angular - HTTP Client

Importamos `Response` para trabajar correctamente con las respuestas HTTP.

```
import { Response } from "@angular/http";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/map";
```

Angular - HTTP Client

```
import { Response } from "@angular/http";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/map";
```

El cliente `Http` trabaja con `Observable`, por tanto debemos importar también este tipo.

Angular - HTTP Client

```
import { Response } from "@angular/http";
import { Observable } from "rxjs/Observable";
import "rxjs/add/operator/map"
```

Se puede extender la funcionalidad de un `Observable` añadiendo operadores. En este caso importamos `map`.

Angular - HTTP Client

4. Ready! Set! Go!

Angular - HTTP Client

```
@Injectable()
export class ProductService {

 getProducts(): Observable<Product[]> {
  return this._http
             .get("api/products")
             .map((data: Response) => data.json() as Product[]);
 }
}
```

Angular - HTTP Client

El servicio `Http` siempre retorna un `Observable` del tipo `T` (genérico).

```
@Injectable()
export class ProductService {

  getProducts(): Observable<Product[]> {
    return this._http
                .get("api/products")
                .map((data: Response) => data.json() as Product[]);
  }
}
```

Angular - HTTP Client

La función `get` del cliente hace una petición con el verbo `GET` sobre el recurso indicado.

```
@Injectable()
export class ProductService {

  getProducts(): Observable<Product[]> {
    return this._http
                .get("api/products")
                .map((data: Response) => data.json() as Product[]);
  }
}
```

Angular - HTTP Client

```
@Injectable()
export class ProductService {

 getProducts(): Observable<Product[]> {
  return this._http
             .get("api/products")
             .map((data: Response) => data.json() as Product[])
 }
}
```

Con `map` convertimos una respuesta genérica `Response` a otro tipo.

Angular - HTTP Client

Con `RequestOptions` configuramos la petición HTTP.

Angular - HTTP Client

Importación

```
import { RequestOptions, Headers, URLSearchParams } from "@angular/http";
```

Angular - HTTP Client

```
getProducts(page: number): Observable<Product[]> {

 let headers = new Headers();
 headers.set("Accept", "application/json");

 let search = new URLSearchParams();
 search.set("page", page.toString());

 let options = new RequestOptions();
 options.headers = headers;
 options.search = search;

 return this._http.get("api/products", options).map(...);
}
```

Angular - HTTP Client

```
getProducts(page: number): Observable<Product[]> {

  let headers = new Headers();
  headers.set("Accept", "application/json");

  let search = new URLSearchParams();
  search.set("page", page.toString());

  let options = new RequestOptions();
  options.headers = headers;
  options.search = search;

  return this._http.get("api/products", options).map(...);
}
```

Con el objeto `Headers` establecemos las cabeceras de la petición.

Angular - HTTP Client

```
getProducts(page: number): Observable<Product[]> {

 let headers = new Headers();
 headers.set("Accept", "application/json");

 let search = new URLSearchParams();
 search.set("page", page.toString());

 let options = new RequestOptions();
 options.headers = headers;
 options.search = search;

 return this._http.get("api/products", options).map(...);
}
```

Con el objeto `URLSearchParams` establecemos los parámetros de consulta.

Angular - HTTP Client

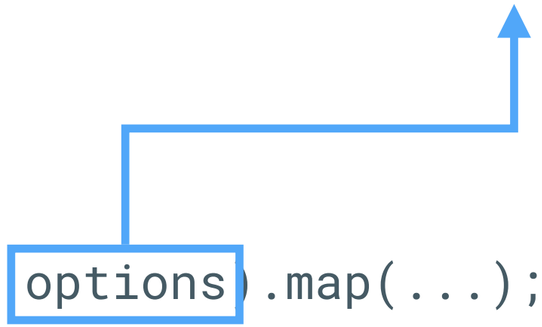
```
getProducts(page: number): Observable<Product[]> {

 let headers = new Headers();
 headers.set("Accept", "application/json");

 let search = new URLSearchParams();
 search.set("page", page.toString());

 let options = new RequestOptions();
 options.headers = headers;
 options.search = search;

 return this._http.get("api/products", options).map(...);
}
```

Con el objeto `RequestOptions` configuramos la petición.

Angular - HTTP Client

```
getProducts(page: number): Observable<Product[]> {

 let headers = new Headers();
 headers.set("Accept", "application/json");

 let search = new URLSearchParams();
 search.set("page", page.toString());

 let options = new RequestOptions();
 options.headers = headers;
 options.search = search;

 return this._http.get("api/products", options).map(...);
}
```

Pasamos el objeto `RequestOptions` en la petición `get` del cliente.



Angular - HTTP Client

Podemos hacer peticiones HTTP con cualquier verbo.

Angular - HTTP Client

Petición POST

```
createProduct(product: Product): Observable<Product> {
   return this._http
               .post("api/products", product)
               .map((data: Response) => data.json() as Product);
}
```

Angular - HTTP Client

Petición POST

```
createProduct(product: Product): Observable<Product> {
   return this._http
              .post("api/products", product)
              .map((data: Response) => data.json() as Product);
}
```

Usando la función `post` hacemos una petición HTTP con el verbo `POST`.
Como segundo parámetro indicamos el cuerpo de la petición.

Angular - HTTP Client

Petición PUT

```
updateProduct(product: Product): Observable<Product> {
   return this._http
               .put(`api/products/${product.id}`, product)
               .map((data: Response) => data.json() as Product);
}
```

Angular - HTTP Client

Petición PUT

```
updateProduct(product: Product): Observable<Product> {
   return this._http
              .put(`api/products/${product.id}`, product)
              .map((data: Response) => data.json() as Product);
}
```

Usando la función `put` hacemos una petición HTTP con el verbo PUT.
Como segundo parámetro indicamos el cuerpo de la petición.

Angular - HTTP Client

Petición DELETE

```
deleteProduct(product: Product): Observable<Product> {
   return this._http.delete(`api/products/${product.id}`);
}
```

Angular - HTTP Client

Petición DELETE

```
deleteProduct(product: Product): Observable<Product> {
   return this._http.delete(`api/products/${product.id}`)
}
```

Usando la función `delete` hacemos una petición HTTP con el verbo `DELETE`.

Angular - HTTP Client

dudas & preguntas

RxJS Observables

Angular - RxJS Observables

- Facilita el trabajo con flujos de datos (streams).
- Proporciona un sistema *Push* de datos.
- Ofrece operadores para transformar, filtrar o combinar flujos.

Angular - RxJS Observables

Importación

```
// Modular
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";
import "rxjs/add/operator/map";

// Complete
import "rxjs/Rx";
```

Angular - RxJS Observables

Importación

```
// Modular
import { Observable } from "rxjs/Observable"
import "rxjs/add/observable/from";
import "rxjs/add/operator/map";

// Complete
import "rxjs/Rx";
```

El paquete `rxjs/Observable` ofrece una versión reducida de la clase `Observable`.

Angular - RxJS Observables

Importación

```
// Modular
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from"
import "rxjs/add/operator/map";

// Complete
import "rxjs/Rx";
```

Desde el paquete `rxjs/add/observable` extendemos los métodos estáticos de la clase `Observable`.

Angular - RxJS Observables

Importación

```
// Modular
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";
import "rxjs/add/operator/map"

// Complete
import "rxjs/Rx";
```

Desde el paquete `rxjs/add/operator` extendemos los operadores de la clase `Observable`.

Angular - RxJS Observables

Importación

```
// Modular
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";
import "rxjs/add/operator/map";

// Complete
import "rxjs/Rx"
```

El paquete `rxjs/Rx` añade todos los métodos estáticos y operadores de la clase `Observable`.

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    observer.next("Hello");
    observer.next("World");
  }
);

observable$.subscribe(console.log);
```

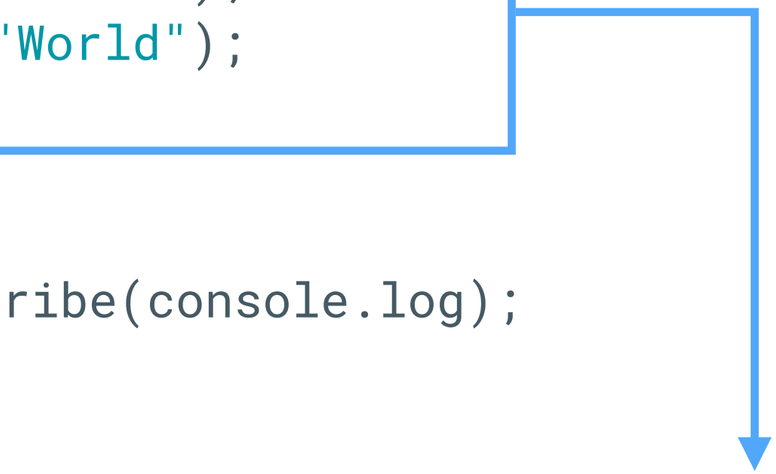
Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
 (observer: Observer<string>) => {
  observer.next("Hello");
  observer.next("World");
 }
);

observable$.subscribe(console.log);
```

Con el método estático `create` creamos un nuevo `Observable`.

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
 (observer: Observer<string>) => {
  observer.next("Hello");
  observer.next("World");
 }
);

observable$.subscribe(console.log);
```



Los valores que produce el `Observable` los consume el `Observer`.

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    observer.next("Hello");
    observer.next("World");
  }
);

observable$.subscribe(console.log);
```

Con `next` notificamos valores a los suscriptores.

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    observer.next("Hello");
    observer.next("World");
  }
);

observable$.subscribe(console.log)
```

Para obtener los valores producidos por el `Observable`, debemos suscribirnos a él con `subscribe`.

Angular - RxJS Observables

¿Qué tipo de información podemos notificar?

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.send("Hello World!");
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.send("Hello World!")
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Con `next` notificamos valores a los suscriptores.

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.send("Hello World!")
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.error("The operation caused an error.");
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.error("The operation caused an error.")
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Con `error` notificamos errores a los suscriptores.

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.error("The operation caused an error.")
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`)
  () => console.info("done")
);
```

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.complete();
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.complete()
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Con `complete` indicamos que el `Observable` no va a producir más valores.

Angular - RxJS Observables

```
let observable$: Observable<string> = Observable.create(
  (observer: Observer<string>) => {
    ...
    observer.complete()
  }
);

observable$.subscribe(
  (data) => console.log(`data: ${data}`),
  (err) => console.error(`error: ${err}`),
  () => console.info("done")
);
```

Angular - RxJS Observables

Jugando con los operadores

Angular - RxJS Observables

```
getProduct(id: number): Observable<Product> {
 this._http
     .get(`api/products/${id}`)
     .map((response: Response) => response.json() as Product)
     .catch(() => Observable.throw("The product does not exist."));
}
```

Angular - RxJS Observables

```
getProduct(id: number): Observable<Product> {
 this._http
      .get(`api/products/${id}`)
      map((response: Response) => response.json() as Product)
      .catch(() => Observable.throw("The product does not exist."));
}
```

Con `map` transformamos un flujo de datos en otro.

Angular - RxJS Observables

```
getProduct(id: number): Observable<Product> {
 this._http
      .get(`api/products/${id}`)
      .map((response: Response) => response.json() as Product)
      catch(() => Observable.throw("The product does not exist."))
}
```

Con `catch` capturamos posibles errores durante el tratamiento del flujo de datos.

Angular - RxJS Observables

```
private _termStream: Subject<string> = new Subject<string>();

items: Observable<string[]> = this._termStream
  .debounceTime(300)
  .distinctUntilChange()
  .switchMap((term: string) => this._wikipediaService.search(term));

onInputChange(term: string) {
 this._termStream.next(term);
}
```

Angular - RxJS Observables

```
private _termStream: Subject<string> = new Subject<string>();

items: Observable<string[]> = this._termStream
  .debounceTime(300)
  .distinctUntilChange()
  .switchMap((term: string) => this._wikipediaService.search(term));

onInputChange(term: string) {
 this._termStream.next(term);
}
```

Con `debounceTime` forzamos una pausa previa a la notificación del dato.

Angular - RxJS Observables

```
private _termStream: Subject<string> = new Subject<string>();

items: Observable<string[]> = this._termStream
  .debounceTime(300)
  .distinctUntilChange()
  .switchMap((term: string) => this._wikipediaService.search(term));

onInputChange(term: string) {
 this._termStream.next(term);
}
```

Con `distinctUntilChange` notificamos un dato solo en caso de diferir de la notificación anterior.

Angular - RxJS Observables

```
private _termStream: Subject<string> = new Subject<string>();

items: Observable<string[]> = this._termStream
  .debounceTime(300)
  .distinctUntilChange()
  .switchMap((term: string) => this._wikipediaService.search(term));

onInputChange(term: string) {
 this._termStream.next(term);
}
```

Con `switchMap` transformamos el flujo de datos, al igual que con `map`, pero solo emitimos el valor más reciente.

Angular - RxJS Observables

dudas & preguntas

Pipes

Angular - Pipes

- Sirven para hacer transformaciones de datos.
- El dato original no se altera.
- Angular ofrece Pipes de serie en el paquete `commons`.

Angular - Pipes

Convención

Documento	`filter-list.pipe.ts`
Clase	`FilterListPipe`

Importación

```
import { Pipe, PipeTransform } from "@angular/core";
```

Angular - Pipes

Creando un Pipe

```
@Pipe({ name: "FilterList" })
export class FilterListPipe implements PipeTransform {
 transform(list: any[], property: string, value: any) {
  return list.filter(item => item[property] === value);
 }
}
```

Angular - Pipes

Creando un Pipe

```
@Pipe({ name: "FilterList" })
export class FilterListPipe implements PipeTransform {
 transform(list: any[], property: string, value: any) {
  return list.filter(item => item[property] === value);
 }
}
```

En `name` establecemos el nombre del Pipe que luego indicaremos en un template para hacer uso del mismo.

Angular - Pipes

Creando un Pipe

```
@Pipe({ name: "FilterList" })
export class FilterListPipe implements PipeTransform {
 transform(list: any[], property: string, value: any) {
  return list.filter(item => item[property] === value);
 }
}
```

Un Pipe tiene un parámetro obligatorio: el dato a transformar.

Angular - Pipes

Creando un Pipe

```
@Pipe({ name: "FilterList" })
export class FilterListPipe implements PipeTransform {
 transform(list: any[], property: string, value: any) {
  return list.filter(item => item[property] === value);
 }
}
```

Puede tener otros parámetros de entrada, tantos como se necesiten.

Angular - Pipes

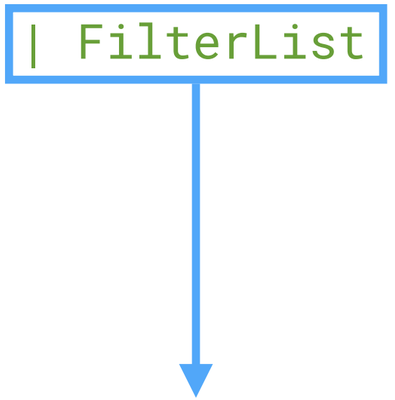
Usando un Pipe

```
<table>
 <tr *ngFor="let user of (users | FilterList:'age':30)">
  <td>{{ user.text }}</td>
 </tr>
</table>
```

Angular - Pipes

Usando un Pipe

```
<table>
 <tr *ngFor="let user of (users | FilterList 'age':30)">
  <td>{{ user.text }}</td>
 </tr>
</table>
```



Con el símbolo | indicamos el uso de un Pipe.

Angular - Pipes

Usando un Pipe

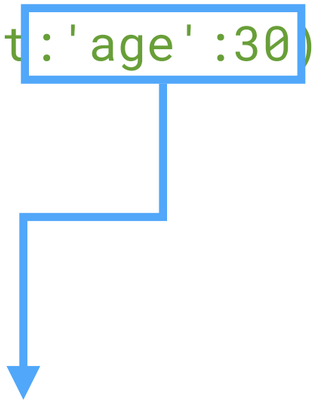
```
<table>
 <tr *ngFor="let user of (users | FilterList:'age':30)">
  <td>{{ user.text }}</td>
 </tr>
</table>
```

El dato que precede al | es el que recibe el formato.

Angular - Pipes

Usando un Pipe

```
<table>
 <tr *ngFor="let user of (users | FilterList:'age':30)">
  <td>{{ user.text }}</td>
 </tr>
</table>
```



Con : indicamos los siguientes parámetros del Pipe.

Angular - Pipes

También podemos usar un `Pipe` desde un `Component`.

Angular - Pipes

```
@Component({ ... })
export class UserComponent {

 constructor(private _datePipe: DatePipe) { }

 formatBirthDate(user: User) {
  return this._datePipe.transform(user.birth, "medium");
 }
}
```

Angular - Pipes

Inyectamos el Pipe como dependencia.

```
@Component({ ... })
export class UserComponent {

  constructor(private _datePipe: DatePipe) { }

  formatBirthDate(user: User) {
    return this._datePipe.transform(user.birth, "medium");
  }
}
```

Angular - Pipes

```
@Component({ ... })
export class UserComponent {

  constructor(private _datePipe: DatePipe) { }

  formatBirthDate(user: User) {
    return this._datePipe.transform(user.birth, "medium")
  }
}
```

A través de la función `transform` hacemos uso del Pipe.

Angular - Pipes

dudas & preguntas

Directives

Angular - Directives

- Facilitan la manipulación del DOM.
- Se apoyan en un potente motor de templates.

Angular - Directives

Existen tres tipos de directivas:

- Components
- Attribute Directives
- Structural Directives

Angular - Directives

Los `Components` ya los conocemos. Veamos los otros tipos.

Attribute Directive

Angular - Attribute Directives

- Cambian el estilo y comportamiento de un elemento.
- Se usan como atributos de los elementos.

Angular - Attribute Directives

Angular ofrece algunas `Attribute Directives` de serie.

Angular - Attribute Directives

`ngClass` directive

```
<div [ngClass]="{
  'sort-asc-icon': ascendingSort,
  'sort-desc-icon': !ascendingSort
}">
  ...
</div>
```

Angular - Attribute Directives

`ngClass` directive

```
<div [ngClass]="{
  'sort-asc-icon': ascendingSort,
  'sort-desc-icon': !ascendingSort
}">
  ...
</div>
```

Con `ngClass` podemos cambiar de manera dinámica las clases CSS asociadas a un elemento.

Angular - Attribute Directives

`ngClass` directive

Establecemos como `key` el nombre de la clase CSS.

```
<div [ngClass]="{
  'sort-asc-icon': ascendingSort,
  'sort-desc-icon': !ascendingSort
}">
  ...
</div>
```

Angular - Attribute Directives

`ngClass` directive

```
<div [ngClass]="{
  'sort-asc-icon': ascendingSort
  'sort-desc-icon': !ascendingSort
}">
  ...
</div>
```

Como `value` indicamos una expresión: si resuelve `true`, la clase CSS se añade al elemento. Por contra, si resuelve `false`, se elimina.

Angular - Attribute Directives

`ngStyle` directive

```
<span [ngStyle]="{
  'text-decoration': isImportant ? 'underline' : 'none'
}">
  ...
</span>
```

Angular - Attribute Directives

`ngStyle` directive

```
<span [ngStyle]="{
  'text-decoration': isImportant ? 'underline' : 'none'
}">
  ...
</span>
```

Con `ngStyle` podemos cambiar de manera dinámica los estilos asociados a un elemento.

Angular - Attribute Directives

`ngStyle` directive

```
<span [ngStyle]="{
  'text-decoration': isImportant ? 'underline' : 'none'
}">
  ...
</span>
```

Establecemos como `key` el nombre del estilo.

Angular - Attribute Directives

`ngStyle` directive

```
<span [ngStyle]="{
  'text-decoration': isImportant ? 'underline' : 'none'
}">
  ...
</span>
```

Como `value` establecemos el valor del estilo,
que puede venir dado como una expresión.

Angular - Attribute Directives

También podemos crear `Attribute Directives` propias.

Angular - Attribute Directives

Convención

Documento	`blink-me.directive.ts`
Clase	`BlinkMeDirective`

Importación

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

Angular - Attribute Directives

1. Inyectar `ElementRef` y `Renderer` como dependencias

Angular - Attribute Directives

```
@Directive({ ... })
export class BlinkMeDirective {

 constructor(
  private _elementRef: ElementRef,
  private _renderer: Renderer) { }
}
```

Angular - Attribute Directives

2. Ready! Aim! Fire!

Angular - Attribute Directives

```
@Directive({ selector: "[myBlinkMe]" })
export class BlinkMeDirective {

 @HostListener("mouseenter") onMouseEnter() {
  this._renderer.setElementClass(
   this._elementRef.nativeElement,
   "blink-animation",
   true
  );
 }
}
```

Angular - Attribute Directives

```
@Directive({ selector: "[myBlinkMe]" })
export class BlinkMeDirective {

 @HostListener("mouseenter") onMouseEnter() {
  this._renderer.setElementClass(
   this._elementRef.nativeElement,
   "blink-animation",
   true
  );
 }
}
```

En `selector` indicamos el atributo HTML que instanciará la directiva.

Angular - Attribute Directives

```
@Directive({ selector: "[myBlinkMe]" })
export class BlinkMeDirective {

 @HostListener("mouseenter") onMouseEnter() {
  this._renderer.setElementClass(
   this._elementRef.nativeElement,
   "blink-animation",
   true
  );
 }
}
```

Es necesario respetar los [ ] del selector. Se recomienda, además, preceder el nombre de la directiva con un texto personalizado para evitar posibles conflictos con otras que tengan el mismo nombre.

Angular - Attribute Directives

```
@Directive({ selector: "[myBlinkMe]" })
export class BlinkMeDirective {

  @HostListener("mouseenter") onMouseEnter() {
    this._renderer.setElementClass(
      this._elementRef.nativeElement,
      "blink-animation",
      true
    );
  }
}
```

Con `@HostListener()` suscribimos la directiva a eventos del elemento que la contiene (host).

Angular - Attribute Directives

```
@Directive({ selector: "[myBlinkMe]" })
export class BlinkMeDirective {

  @HostListener("mouseenter") onMouseEnter() {
    this._renderer.setElementClass(
      this._elementRef.nativeElement,
      "blink-animation",
      true
    );
  }
}
```

`Renderer` es el encargado de manipular el estilo de los elementos.

Angular - Attribute Directives

```
@Directive({ selector: "[myBlinkMe]" })
export class BlinkMeDirective {

 @HostListener("mouseenter") onMouseEnter() {
  this._renderer.setElementClass(
   this._elementRef.nativeElement
   "blink-animation",
   true
  );
 }
}
```

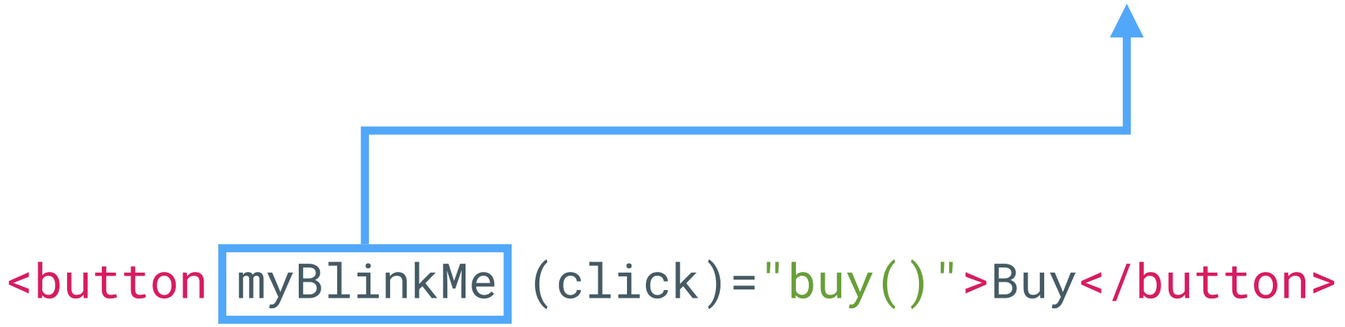
`ElementRef` es una referencia al elemento donde se ha instanciado la directiva. Usamos su propiedad `nativeElement` para interactuar con él.

Angular - Attribute Directives

```
<button myBlinkMe (click)="buy()">Buy</button>
```

Angular - Attribute Directives

Establecemos la directiva como atributo de un elemento para instanciarla en él.



Structural Directive

Angular - Structural Directives

- Cambian la estructura de un elemento.
- Se apoyan en el elemento `<template>` de HTML5.
- Ofrecen el *syntactic sugar* `*` para facilitar su uso.

Angular - Structural Directives

```
<!-- Standard -->
<template [ngIf]="product.available">
  <button>Buy</button>
</template>

<!-- Using the syntactic sugar -->
<button *ngIf="product.available">Buy</button>
```

Angular - Structural Directives

Una `Structural Directive` debe usarse sobre un elemento `<template>`.

```
<!-- Standard -->
<template [ngIf]="product.available">
  <button>Buy</button>
</template>

<!-- Using the syntactic sugar -->
<button *ngIf="product.available">Buy</button>
```

Angular - Structural Directives

Angular puede manipular el DOM correspondiente al contenido del elemento `<template>`.

```
<!-- Standard -->
<template [ngIf]="product.available">
 <button>Buy</button>
</template>

<!-- Using the syntactic sugar -->
<button *ngIf="product.available">Buy</button>
```

Angular - Structural Directives

```
<!-- Standard -->
<template [ngIf]="product.available">
  <button>Buy</button>
</template>

<!-- Using the syntactic sugar -->
<button *ngIf="product.available">Buy</button>
```

Haciendo uso del `*` evitamos definir el elemento `<template>`.

Angular - Structural Directives

Angular ofrece algunas `Structural Directives` de serie.

Angular - Structural Directives

`ngIf` directive

```
<button *ngIf="product.available">Buy</button>
```

Angular - Structural Directives

`ngIf` directive

```
<button *ngIf="product.available">Buy</button>
```

Con `ngIf` indicamos si escribir o no un elemento en el DOM.

Angular - Structural Directives

`ngIf` directive

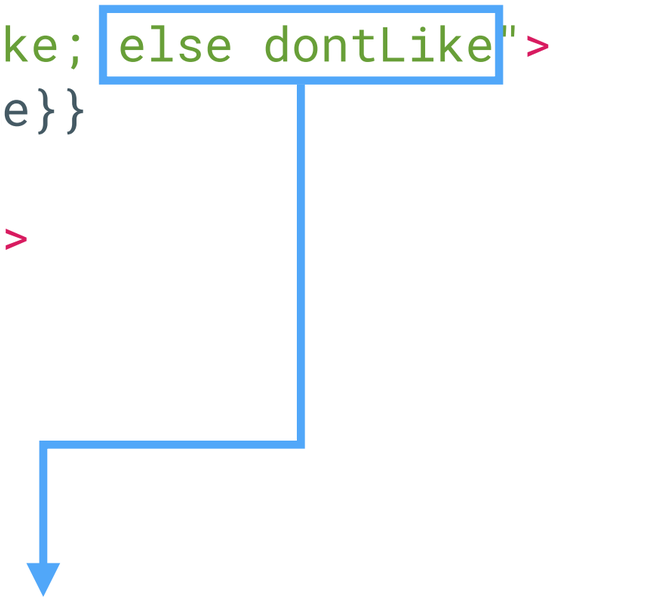
```
<button *ngIf="product.available">Buy</button>
```

Si la expresión indicada resuelve `true`, el elemento se escribe en el DOM. En caso de resolver `false`, se elimina.

Angular - Structural Directives

`ngIf` directive - else

```
<div *ngIf="product.like; else dontLike">
  I like {{product.name}}
</div>
<ng-template #dontLike>
  :(
</ng-template>
```

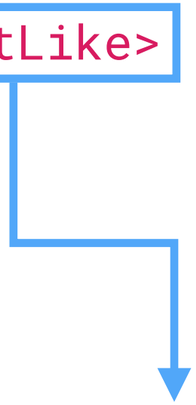


Si la expresión indicada resuelve `false`, le decimos que template usar

Angular - Structural Directives

ngIf directive - else

```
<div *ngIf="product.like; else dontLike">
  I like {{product.name}}
</div>
<ng-template #dontLike>
  :(
</ng-template>
```



Si la expresión indicada resuelve `false`, se evalúa el template indicado

Angular - Structural Directives

ngFor directive

```
<ul>
 <li *ngFor="let product of products">
  {{ product.name }}
 </li>
</ul>
```

Angular - Structural Directives

ngFor directive

```
<ul>
  <li *ngFor="let product of products">
    {{ product.name }}
  </li>
</ul>
```

Con ngFor escribimos una colección de elementos en el DOM.

Angular - Structural Directives

ngFor directive

```
<ul>
  <li *ngFor="let product of products">
    {{ product.name }}
  </li>
</ul>
```

Con `let` asignamos en una variable local el elemento de la colección indicada por `of` por el cual se está iterando.

Angular - Structural Directives

ngSwitch directive

```
<div [ngSwitch]="order.status">
  <span *ngSwitchCase="'received'">Received</span>
  <span *ngSwitchCase="'sent'">Sent</span>
  <span *ngSwitchDefault>In process</span>
</div>
```

Angular - Structural Directives

ngSwitch directive

```
<div [ngSwitch]="order.status">
  <span *ngSwitchCase="'received'">Received</span>
  <span *ngSwitchCase="'sent'">Sent</span>
  <span *ngSwitchDefault>In process</span>
</div>
```

Con ngSwitch observamos el valor de una expresión y escribimos en el DOM el elemento que corresponda.

Angular - Structural Directives

ngSwitch directive

```
<div [ngSwitch]="order.status">
  <span *ngSwitchCase="'received'">Received</span>
  <span *ngSwitchCase="'sent'">Sent</span>
  <span *ngSwitchDefault>In process</span>
</div>
```

Escribimos tantos ngSwitchCase como posibles valores necesitemos controlar. En caso de coincidencia, el elemento se escribe en el DOM.

Angular - Structural Directives

ngSwitch directive

```
<div [ngSwitch]="order.status">
  <span *ngSwitchCase="'received'">Received</span>
  <span *ngSwitchCase="'sent'">Sent</span>
  <span *ngSwitchDefault>In process</span>
</div>
```

Definimos ngSwitchDefault para capturar todos aquellos casos que escapen a los ngSwitchCase indicados.

Angular - Structural Directives

También podemos crear `Structural Directives` propias.

Angular - Structural Directives

Convención

Documento	`delay-me.directive.ts`
Clase	`DelayMeDirective`

Importación

```
import { Directive, TemplateRef, ViewContainerRef } from "@angular/core";
```

Angular - Structural Directives

1. Inyectar `TemplateRef` y `ViewContainerRef`

Angular - Structural Directives

```
@Directive({ ... })
export class DelayMeDirective {

 constructor(
  private _templateRef: TemplateRef<any>,
  private _viewContainerRef: ViewContainerRef) { }
}
```

Angular - Structural Directives

`TemplateRef` se apoya en genéricos; indicamos el tipo `any`.

```
@Directive({ ... })
export class DelayMeDirective {

 constructor(
  private _templateRef: TemplateRef<any>
  private _viewContainerRef: ViewContainerRef) { }
}
```

Angular - Structural Directives

2. Simon says go!

Angular - Structural Directives

```
@Directive({ selector: "[myDelayMe]" })
export class DelayMeDirective {

 @Input("myDelayMe") set delay(ms: number) {
  setTimeout(() => {
   this._viewContainerRef
       .createEmbeddedView(this._templateRef);
  }, ms);
 }
}
```

Angular - Structural Directives

```
@Directive({ selector: "[myDelayMe]" })
export class DelayMeDirective {

  @Input("myDelayMe") set delay(ms: number) {
    setTimeout(() => {
      this._viewContainerRef
          .createEmbeddedView(this._templateRef);
    }, ms);
  }
}
```

Podemos enlazar valores en la directiva usando `@Input()`.

Angular - Structural Directives

```
@Directive({ selector: "[myDelayMe]" })
export class DelayMeDirective {

  @Input("myDelayMe") set delay(ms: number) {
    setTimeout(() => {
      this._viewContainerRef
          .createEmbeddedView(this._templateRef);
    }, ms);
  }
}
```

`ViewContainerRef` es una referencia al marco que contiene la vista de la directiva; lo usaremos para manipular el DOM contenido.

Angular - Structural Directives

```
@Directive({ selector: "[myDelayMe]" })
export class DelayMeDirective {

 @Input("myDelayMe") set delay(ms: number) {
  setTimeout(() => {
   this._viewContainerRef
       .createEmbeddedView(this._templateRef);
  }, ms);
 }
}
```

`TemplateRef` es una referencia a la vista de la directiva; es decir, al elemento `<template>` indicado.

Angular - Directives

dudas & preguntas

GRACIAS

www.keepcoding.io